Low Power Design

Design Techniques for Hardware and Firmware Engineers

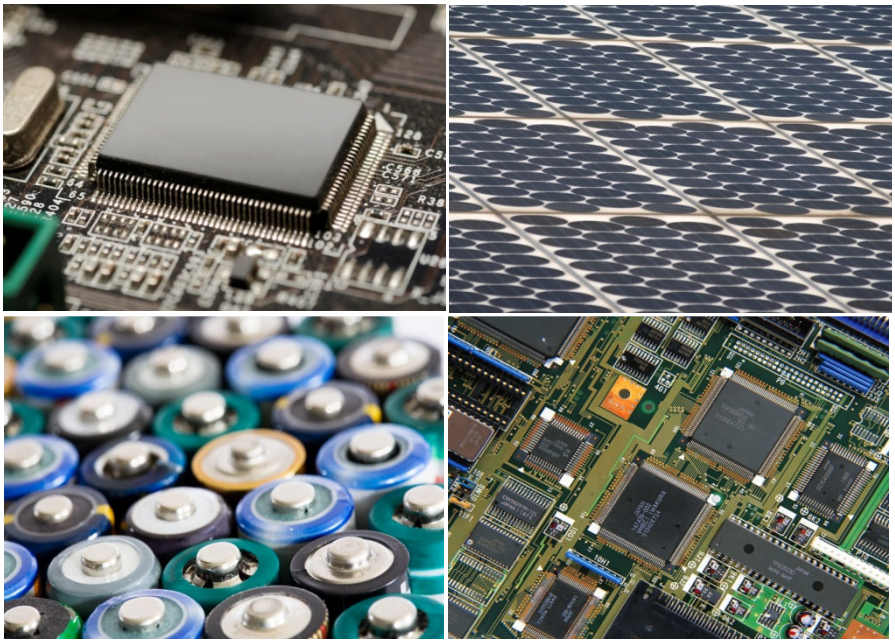


Table of Contents

Cover photo credits:
Micro close-up image, © Leolintang | Dreamstime Stock Photos & Stock Free Images
Solar panels image, © Gyuszko | Dreamstime Stock Photos & Stock Free Images
Batteries image, © Gbp | Dreamstime Stock Photos & Stock Free Images
Circuit board image, © Dlavrmoney | Dreamstime Stock Photos & Stock Free Images

Introduction

We live in an era where saving energy is a topic that receives considerable attention. For design engineers, there are many different motivations for saving energy from your company being able to claim your newest gizmo has longer battery life than any of its competitors to reducing the air pollution and greenhouse gas emissions associated with generating electricity to legal or market requirements. Until recently, decreasing energy usage in electronic circuit designs was primarily a concern for designers of battery power products. Today, legislative restrictions on energy usage are covering more and more types of products and the marketplace pressures for reducing energy usage are impacting nearly all types of electronic products from consumer electronics to IT equipment and even industrial and manufacturing equipment. In our "instant on" world where few things are ever really turned completely off, the energy a device uses when it is doing nothing is becoming more important than the energy it uses while operating. These so called phantom loads drive megawatts of electricity generation daily and are squarely in the sights of legislators writing energy efficiency regulations.

There are a number of ways to reduce the energy usage of your designs. Your hardware design will establish a baseline for the minimum amount of power your product consumes but for many types of products the firmware will really determine the overall power usage. Sophisticated software power management schemes that were once mainly employed in laptop computers and cell phones are being implemented in increasingly less complex products. However, simply selecting the latest low power micro and implementing sophisticated power management is the "big stick" approach to power savings. There are dozens of hardware design decisions and hundreds of firmware design decisions that can either enhance or sabotage your power saving efforts.

There are many things to consider for your design, some will have big pay-offs in power savings but the accumulation of the small savings can sometimes be more significant. Many of the things presented here can be implemented in existing designs, particularly the firmware techniques. The over-riding theme throughout this book is anything your design does that it doesn't need to do is wasting energy. That sounds very obvious but many of the things we do in our designs that waste energy often are not obvious. This is particularly true for firmware driven designs but even something as simple as driving an LED at 4mA instead of 5mA can provide significant power savings if that LED is on most of the time.

Early in my career, high performance was the main feature of the products I developed. It may seem counterintuitive but many firmware techniques for high performance are also well suited for low power since minimizing the time required for a specific task also reduces the amount of power used to perform that task. I designed my first battery powered product in 1995, it was essentially a laptop computer embedded in a telephony test instrument. Most of the information presented here wasn't published anywhere at the time (some of it was learned during that development). Low power usage has been an important aspect of most of my design work for the past decade plus. In some cases the primary motivation was to minimize the heat generated within sealed embedded control systems with no airflow, for another the motivation was to reduce the size (and cost) of the solar panel and

lead-acid battery to power an industrial control system. Most recently, the focus has been on maximizing battery life on two projects, one a wireless sensor for a sports medicine application and the other for a remote monitoring system where replacing batteries for one unit could literally cost upwards of ten thousand dollars. During each of these projects I learned something new about low-power design and much of that learning has been captured here. For the sake of completeness, I've also included many of the standard do's and don'ts you will find in articles and app notes on low power design.

This book does not recommend any particular micros or even one micro architecture over another. Instead, my goal was to point out the types of things that need to be considered when selecting a micro. There are simply too many variables from application to application that could make a specific part power efficient in one application but very inefficient in another application. Beyond selecting a micro, I also wanted to provide some tips and techniques to help with your power savings efforts and point out some traps that need to be avoided. Whether you are a hardware engineer, a firmware engineer or have to wear both hats, there are many ways you can save power in your designs. Hopefully you can apply much of the information included here and incorporate many of these techniques in your designs, whether they are hardware or firmware.

Micro Selection

For many engineers, the basis for the micro selection for a new product design is either use the same micro as on previous products or pick the latest micro from your favorite micro manufacturer. While there are many good reasons to work within the same family of micros, if low power usage has recently become a requirement for your products it may be time to evaluate other micros. Particularly for firmware intensive products it pays to do your research up front and select a micro that will allow you to have a power efficient design. You must go way past the operating and sleep mode current specs in the datasheet and research the low power modes and the peripherals a part has to make sure you can use those peripherals in the power savings modes you need to implement.

8-bit vs 32-bit Micros

It seems like a no-brainer that an 8-bit micro would inherently use less power than a 32-bit micro. This isn't necessarily true and for most applications just looking at the data sheet current specs won't provide an accurate comparison. Most 8-bit micros have much lower sleep mode currents than 32-bit micros of similar vintage but a 32-bit micro is likely to spend considerably more time sleeping than an 8-bit micro because it can perform its tasks much faster. In applications with complex algorithms or floating point math, a 32-bit micro may be considerably more power efficient simply because it can complete tasks in considerably fewer clock cycles than an 8-bit micro (floating point operations can take thousands of clock cycles on many 8-bit micros). Even in simpler applications, for firmware written in C the math required to calculate addresses for structures or arrays can greatly increase the power consumption of an 8-bit micro for a given task compared to a 32-bit micro.

There are also some 16-bit micros that shouldn't be ignored, particularly if an 8-bit micro is suitable for the task and your firmware is in C. If your firmware fits in a 64KB address space then a 16-bit micro can greatly reduce the number of instructions required for calculating structure and array addresses. The main drawback to 16-bit micros in general is most of them are older architectures and won't have many of the power management features implemented in the newer generations of 8-bit and 32-bit micros.

Many of the micros released within the past 2-3 years have operating currents 1,000X to 10,000X higher than their sleep and deep sleep mode current. The upcoming generation of low power micros will increase this difference by another 10X or more. For the ultimate power savings it is crucial that the micro spend as little time executing code and the most time in a sleep or deep sleep mode. If your firmware will be written in C, for anything but the simplest applications 32-bit micros will have a significant advantage in executing code quickly. Consider the equation below for the current used during a specific event:

$$I_{event} = (Time_{operating} \times I_{operating}) + (Time_{sleep} \times I_{sleep})$$

Assume a hypothetical 8-bit micro with a 1mA operating current and 50uA sleep current and a 32-bit micro with twice the current levels (2mA and 100uA). The micro must perform a specific operation that includes 100mS of sleep time while waiting on a mechanical action. Assuming the 32-bit micro is

able to perform the task 4X faster than the 8-bit micro, the two equations below show the current required is cut by over one third with the 32-bit micro.

8-bit micro => (50 x 0.001) + (100 x 0.00005) = 0.055
32-bit micro => (12.5 x 0.002) + (100 x 0.0001) = 0.035

Architecture

It would be hard to say any one particular microcontroller architecture has lower power consumption than another for a given application. An architecture with single clock cycle instruction execution would use less power for a given task than a micro that required multiple clock cycles per instruction given that everything else is equal (which is rarely the case). For applications that are very data intensive, a Harvard architecture with separate data paths for program and data memory may use less power than a micro with a shared data path for program and data memory because it can complete tasks faster. On the other hand, for an application that has very little data memory use, the shared data path architecture could be lower power than a Harvard architecture simply because there is less circuitry to power.

It would be fair to say that a more modern architecture will likely provide better control over power consumption. A more modern architecture may allow clock and power control for individual peripherals instead of the all or none approach that older micros used for power management. This can be crucial for power savings in applications that utilize a number of on-chip peripherals like A/D converters, UARTS, DMA controllers, etc.

Program Location

For most microcontrollers, accessing on-chip RAM typically use less power than accessing on-chip Flash so you may be able to save power by executing code in RAM instead of Flash. A number of micros don't support executing code in RAM (particularly Harvard architecture micros) and the power consumption when accessing RAM compared to Flash varies so be sure to research this in depth before counting on this power savings. This may also not be true if the Flash and RAM data busses aren't the same width (a 16-bit path to Flash and 8-bit path to RAM for instance).

You should avoid using off-chip memory for program execution if at all possible. Besides taking more clock cycles to access off-chip memory, the power consumed by the off-chip I/O buffers and the external memory devices make off-chip memory accesses very expensive in terms of power usage.

C or Assembly Language

It is hard to beat lovingly hand crafted assembly language for efficient power usage. However, unless the energy usage for your application is so critical you are counting nanoamps, it is hard to justify the extra development time and maintenance issues associated with assembly language.

Most modern micros (even some 8-bit parts) are architected to work efficiently with compiled C code. If a micro has a very limited/fixed stack size, doesn't support indexed memory addressing modes, can't do arithmetic or logical operations directly on memory locations or doesn't deal well

with immediate addresses or values, it won't efficiently execute C code and will waste considerable power because of it.

Clock Frequency

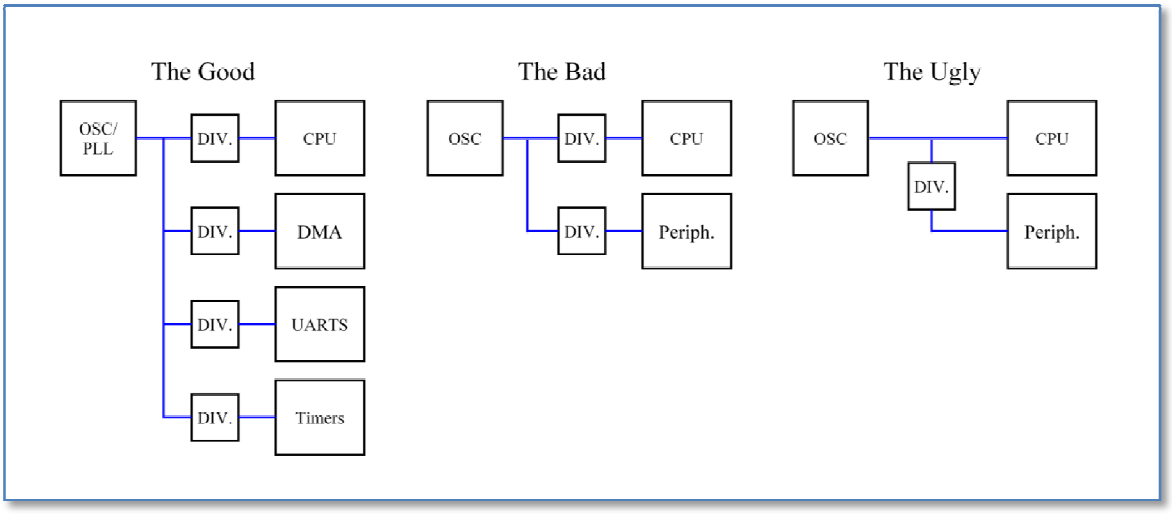
Most modern micros are implemented in a CMOS process where power consumption scales almost linearly with clock frequency. Consider again the equation for current used by a micro for performing a particular task:

$$I_{event} = (Time_{operating} \times I_{operating}) + (Time_{sleep} \times I_{sleep})$$

Don't fall into the trap of thinking you can run the micro faster to use less current. Since the power scales linearly with the clock frequency, running twice as fast while using twice as much current produces the same number for the operating portion of the equation. Ironically, if the task occurs repeatedly at a specific interval, the total current used will actually increase since the sleep time increases.

A number of modern micros use a PLL to generate a higher speed clock for the micro core. Before using one of these parts, if you plan to use clock throttling as part of your power management scheme you must be aware that the PLL response time when turning it on/off or even significantly changing the clock speed can greatly increase the firmware's response time for events that trigger the clock speed increase.

It is also important to consider the internal clock chains for anything but the simplest micros. As shown in the diagram below, the more branches on the clock tree the more control you generally have over clock speeds and enables. In simpler micros the peripheral clock is divided down from the micro core clock. Two things to keep in mind in this case are (1) the micro must be running when any peripherals are being used and (2) reducing the micro clock speed is not an option for saving power when a higher speed clock is required for a peripheral (like a high-speed UART). Most modern micros will have at a minimum a clock for the micro core and another clock for the on-chip peripherals. This isn't sufficient if your goal is the ultimate power savings.



Regarding UARTs, some of the more recent micros provide a fractional divider in addition to the basic baud rate divider for better accuracy at higher baud rates. The drawback to this capability is it usually requires a clock of at least 16X the desired baud rate so a 115K baud rate requires a clock of over 1.8Mhz, greatly increasing the power used by the UART. There is a family of Cortex M3 based micros that have the fractional divider that requires the 16X clock whether the fractional divider is being used or not so this can be an expensive feature power-wise even when not used.

Internal vs External Peripherals

With modern micro architectures, using internal peripherals tends to be considerably more power efficient than using external peripherals. There are a number of reasons for this:

- The main reason is most modern micros operate with a lower core voltage so internal peripherals operate at the lower core voltage and external peripherals operate at the higher system voltage.
- The trend towards using 2 and 3 wire peripheral interfaces like I2C and SPI to reduce pin count and package size means clocked serializer and deserializer circuits are required for external peripherals while internal peripherals have much faster parallel interfaces. These added circuits consume additional power plus the micro is still using power while waiting for data transfers with the external device to complete.
- Passing signals through the micro's I/O buffers to access off-chip peripherals consumes additional power. When you must use off-chip peripherals, high-speed serial SPI/I2C interfaces may be more efficient than 8 or 16 bit parallel interfaces since the I/O buffers will always consume power, not just during the data transfers. Using these serial interfaces will be even more power efficient on micros that have SPI/I2C controllers so the micro can sleep while the transfers take place. If you must use parallel interface external peripherals, the I/O pins used for data lines should be driven low when not in use to minimize power.

GPIO

There are a number of ways that GPIO can impact power consumption:

- The I/O ports on most older micro architectures have fixed drive strength. Many modern architectures provide drive strength control on a per port or even per pin basis so the drive strength can be tailored to the circuit requirements.
- Similarly, if your design has more than a few GPIO outputs that change states frequently, for ultimate power savings look for a micro that supports programmable slew rates. Faster signal transitions require more power than slower ones for the same capacitive load.
- Most micros provide options for internal pull-ups and/or pull-downs on their I/O ports. While convenient to use and a good way to reduce total parts count, these internal pull-ups tend to not be well controlled and can be as low as a few K-ohms on some micros, leading to excessive current draw. Consider a 3.3V micro, with an internal 5K pull-up on a switch input that is normally grounded. This 5K pull-up will pull 660uA, compared to 33uA for an external 100K pull-up. Some micros also provide control of the pull-ups/pull-downs on a per port basis so all inputs on a port will have the pull-up/pull-down enabled just because one input needs it. If extreme low

power is required, it's much better to use discrete pull-up/down resistors only where needed and with as high a value resistor as can be tolerated in the circuit.

- Unless the micro datasheet says otherwise, it's usually best for low power usage to configure unused GPIO as outputs driving a logic low. In this configuration, the I/O pin is trying to sink current so it is using virtually no power.
- Slowly rising or falling input signals can cause excessive power consumption and even generate noisy oscillations while the input is between logic low and logic high thresholds. Few micros provide Schmitt trigger inputs so an external Schmitt trigger buffer may be needed to provide fast, clean transitions to the micro. If the slow transition time is due to a weak pull-up it may be more efficient to use a stronger pull-up than to power the Schmitt trigger device.

Low Power Modes

This can get very confusing because you can run into terms like "idle", "nap", "snooze", "sleep", "hibernate" and "deep sleep". There are no standards regarding low-power states and the power levels and functionality available for a given mode name may vary widely from micro to micro. There are several things you need to pay close attention to regarding power saving modes:

- Make sure in the low power modes you want to implement that the on-chip peripherals you need are functional. For example, in some micros the UARTs are fully operational while the micro is sleeping, others require the micro to be running for the UART to be operational.
- One popular family of small micros powers down its main registers and internal SRAM in deep-sleep mode. Only the instruction pointer and two special registers for program state information are maintained while in deep-sleep. Other than these two registers, waking up from deep sleep isn't much different than coming out of reset. One very important difference is any variables that the compiler caused to be automatically initialized on reset or are initialized in your reset code won't be initialized when coming out of the deep sleep mode. Your wake-up code must handle initializing these variables, possibly delaying the response to the event that triggered the wake-up.
- Pay close attention to what conditions bring a micro out of its low power states. In light and medium sleep modes, interrupts from on-chip peripherals and external interrupts will generally wake the micro. The lowest current modes usually have the fewest options for waking the micro. Some micros allow GPIO pins to be configured to wake the micro on a per-pin or per-port basis while some micros will only wake on transitions on external interrupt pins or a specific "wake" pin.
- If you select a micro that turns off or reduces its core voltage in a low-power mode, be careful with the amount of capacitance on core voltage pins. These pins typically only require a filter capacitor so the smallest capacitor possible should be used since it will be charged and discharged every time the low-power mode is entered or exited.

Circuitry

Logic families

Obviously, not all logic families were created equal or there wouldn't be such a wide variety of them. For the most part, any of the low-voltage and low-power CMOS logic families will perform well in low power circuits. There are a few things to pay attention to when using any logic devices:

- CMOS logic likes its inputs to be near ground or the supply voltage rail. The inputs to CMOS devices are often spec'd below Vcc * 0.3 for a logic low and above Vcc * 0.7 for a logic high. In between these levels is a "no man's land" you must avoid. Driving a CMOS device input with a driver that doesn't drive a logic high close to the supply voltage (such as a "TTL" output) generally works but uses more power than it needs to and may cause oscillations. If you must mix CMOS and TTL logic levels consider using HCT/AHCT devices that are designed for TTL level inputs.
- CMOS logic in particular does not like floating inputs, they consume more power than a logic high or low input and will often cause oscillations. For unused gates, tie the inputs to either ground or the power supply rail so that the output is low in order to use the least amount of power.
- The comments about slow rise/fall times under the "GPIO" in the "Micro Selection" section also apply to CMOS logic devices.

Single vs. Multi-gate Packages

If you only require one or two gates out of a multi-gate part, consider using a lower gate count version of the same part for ultra-low power. Even though CMOS power consumption is primarily caused by signal level changes, the quiescent power for the unused gates in a part adds up and can be considerable in a system that uses very low current power sources like energy harvesting.

Enables

When using external devices that have enable pins, take advantage of them to reduce power. For example, if you have an analog input going into an op-amp in front of an A/D converter, unless you disable it that op-amp is always performing its task and needlessly using power. Depending on the part, that can be from tens of microamps to several milliamps of wasted power. Be sure to check that the enable pin is actually a device enable and not just an output enable.

In some cases you can make your own enables. If you monitor temperature with an NTC/PTC or monitor voltages with a resistor divider, these parts are using power all the time. Using a FET to turn the voltage off to these parts when not in use can save that power. If extreme accuracy isn't required, in some cases you can simply use a GPIO to drive one side of the circuit. You can also take the GPIO to an A/D input so the measurements can be scaled based on the actual GPIO voltage for more accurate measurements using slightly more power.

LEDs

There is no doubt LEDs can provide important feedback to users but they can also be huge power wasters. Where many of the things discussed here can save or cost microamps, LEDs can waste milliamps of precious battery capacity. Several things to consider for LEDs:

- Consider the brightness you really need for an effective indicator. Many products have LEDs that are much brighter than they really need to be. Unless your product is used outdoors, used in a high ambient light situation or there is a big gap between the LED and a light pipe you may be able to reduce the current and still have an effective indicator. You may also be able to run a high-brightness LED at a lower current than a standard LED for comparable brightness.
- Use a flashing LED instead of a constantly on indication. Even a 50/50 duty cycle will save 50% compared to always on. In most cases considerably lower duty cycles can be used and may be preferable to a distracting fast flash. If you need a fast flash, with most LEDs turning them on for just a few milliseconds produces a visible flash.
- The light from an LED persists for a relatively long time after the LED is turned off. If an LED must be constantly on, drive it with a PWM so it gives the appearance of being constantly on. You can get into trade-offs between PWM duty cycle and LED brightness but if you pulse the LED frequently enough you won't lose much brightness while saving considerable power.

The value of pull-ups

Be careful when selecting the values for pull-up and pull-down resistors. This is particularly important for pull-up/down resistors on switches and signals that are normally in an active state (assuming the pull-up/down is intended to pull the signal inactive). A 10K pull-up to 3.3V on a switch that is grounded will pull 330uA. A 100K pull-up in the same circuit will "only" pull 33uA but considering micros with sub-milliamp sleep currents that can be a significant percentage of the total current draw. Don't forget to multiply that current by the number of switches you have, it adds up quickly.

Power managed devices

If your design requires a complex part such as an external A/D converter or a MEMs accelerometer, you may be able to select a device that has power management features. It may have an enable pin as discussed earlier, a sleep bit in a control register or the part may put itself in a lower power mode due to inactivity. Be careful in selecting such a part because it may require some amount of time when exiting the lower power mode before it is operational and that time may be buried in the datasheet text and not called out in the specs. For certain types of MEMs devices this wake-up time can take several milliseconds.

Sensor devices such as MEMs accelerometers are becoming more intelligent and enabling power savings by off-loading functions from the micro. Look for parts that have features such as threshold detection that generates an interrupt to the micro when a programmed threshold is crossed. This type of feature doesn't require much extra power for the device and can save considerable power on the micro since it can sleep until the threshold crossing interrupt is received instead of polling the device. Some multi-channel A/D converters can take and store a series of readings on selected channels before generating an interrupt to the micro.

Power Supplies

This discussion assumes your product is either battery powered or you have already selected the AC/DC power supply if AC line powered. The discussions on voltage regulators are in regard to any DC/DC regulators for generating voltages below your main power rail.

Lowest common voltage

Reducing clock frequency has a near linear reduction in power but reducing voltage provides a squared reduction. This can be illustrated using Ohm's law and some basic algebra:

Current = Voltage / Resistance

Power = Voltage x Current

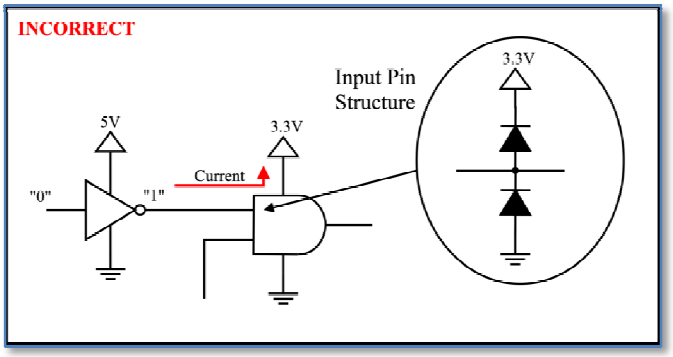
Power = (Voltage x Voltage) / Resistance

This makes it important to use the lowest voltage possible for the main supply rail. However, you must also pay special attention to your system environment before selecting your system voltage level since lower voltages mean reduced noise margin.

Split power rails

If ultra-low power is important and you have devices that can operate at a lower voltage than the main supply rail, it may make sense to add a voltage regulator and even level translators if needed. It may also make sense to power a circuit that interfaces to the external world with a higher voltage for improved noise immunity and use a lower voltage rail for the remainder of your circuits. If your design has split power rails, there a few things to keep in mind, particularly if you decide not to use level translators between the voltage domains:

- Driving a logic high from a higher voltage domain to a lower voltage domain should be done with care. Some parts (primarily micros) may have 5V tolerant inputs but even most parts with a wide operating voltage range still restrict their inputs to slightly above the Vcc rail. As shown below, the ESD protection diode in the lower voltage part MAY prevent over-voltage damage to that device but the excess voltage will be shunted to the lower voltage rail and the wasted power will be turned into heat, possibly destroying the part anyway. A series resistor help prevent damage to the lower voltage part but will still waste power.



- Conversely, driving a logic high from a lower voltage domain to a higher voltage domain may work unless there is a big difference in voltage rails like a 2.5V device driving a 5V device. In this case the high output from the lower voltage device won't reach the high level input threshold of the higher voltage part. If the circuit does work, it will use more power than expected since the input is between the logic low and high thresholds.

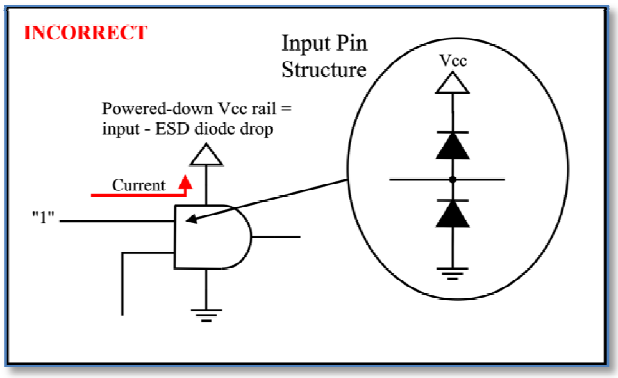
Regulator simplicity versus efficiency

Linear regulators are often used because they are usually cheaper, smaller and easier to design in than switching regulator circuits. The downside to a linear regulator is it generally isn't as efficient as a switching regulator circuit, particularly across a wide load range. All voltage regulators tend to be most efficient near their rated loads. Most new switching regulators intended for low power designs implement some variation of a burst mode for better efficiency at low currents. If your circuit is normally in a low-power state, make sure the regulator you choose (linear or switching) has reasonably good efficiency at that power level. If you are after the ultimate power savings and your product has a significant difference in current draw between operating and inactive modes it may be beneficial to use one regulator for operating mode and another for inactive mode (but make sure both regulators can handle a voltage present on their output when they are disabled).

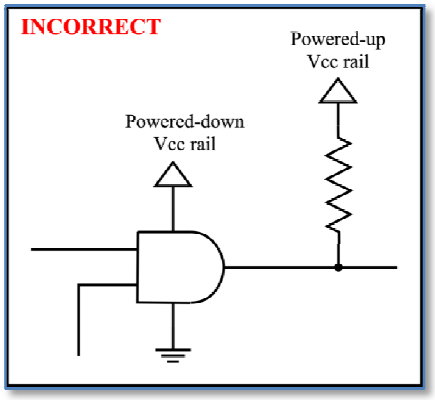
Powering Down Circuits

Powering down circuits that aren't in use provides the ultimate power savings but isn't always as easy as it sounds. Several things to consider are:

- Inputs to powered-down circuits must be driven low, otherwise the ESD protection diode in the input to a powered down device provides a path to power-up the device (see diagram below). With a high drive output and low-power circuit the circuit may be fully functional even when you think it is powered down. With a low drive output and higher power circuit the driving IC may be destroyed. Both scenarios may be missed in the lab and result in failures in the field.



- Make sure that outputs from the circuit to be powered down don't have pull-ups to a different voltage rail as shown in the diagram on the following page. If any outputs from the powered down circuit go directly to a micro make sure the micro's internal pull-up is disabled when the circuit is turned off.



- It is important to use the minimum amount of capacitance on any voltage rails that will be powered down. Each time the circuit is turned on/off, this capacitance will be charged and discharged, potentially wasting a considerable amount of power.
- Circuitry that has been turned "off" may continue to operate for some period of time as the voltage rail drops. This is another reason to minimize the capacitance on these voltage rails.
- Circuitry that has been turned "off" may operate erratically during the time the voltage is dropping. To help prevent this, it can be beneficial to connect an N-channel FET from the voltage rail through a resistor to ground to quickly bring the voltage rail to 0V. Depending on the amount of capacitance on the voltage rail, the resistor could have several volts across it for a long enough period of time to heat up the resistor. Make sure to use a resistor with a high enough wattage rating to handle this.

Batteries

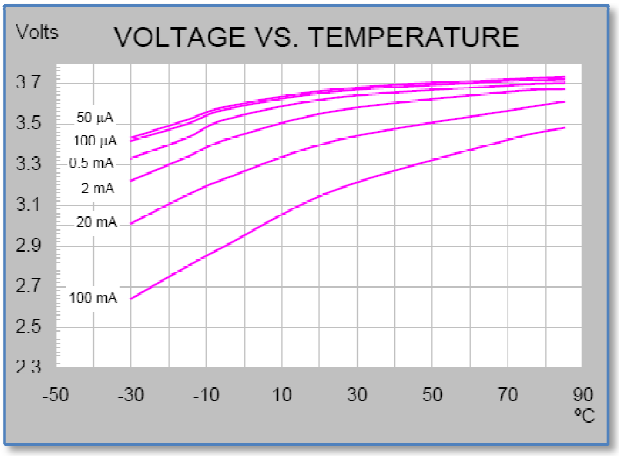
There are a wide variety of battery technologies available today, ranging from the traditional lead-acid battery to the latest lithium polymer (LiPo) batteries. Batteries seem like they should be easy to use, just hook them up and go. In fact you can usually do just that but chances are you won't be satisfied with the resulting charge duration or battery life. Batteries can easily be abused if you don't fully understand their specs. Some battery chemistries are more tolerant of being abused than others but any abuse tends to shorten battery life. Each battery technology has its own unique set of characteristics and many of those characteristics vary based on the environment they are in, how they are used and for rechargeable batteries, how they are charged. To a large degree and for the sake of this discussion, you can model a battery as a fixed voltage source connected to the load through a variable series resistor. There are a number of things such as temperature, load and age that act on this variable resistor to decrease the battery output.

Below are several important things you should know that apply to nearly all battery chemistries.

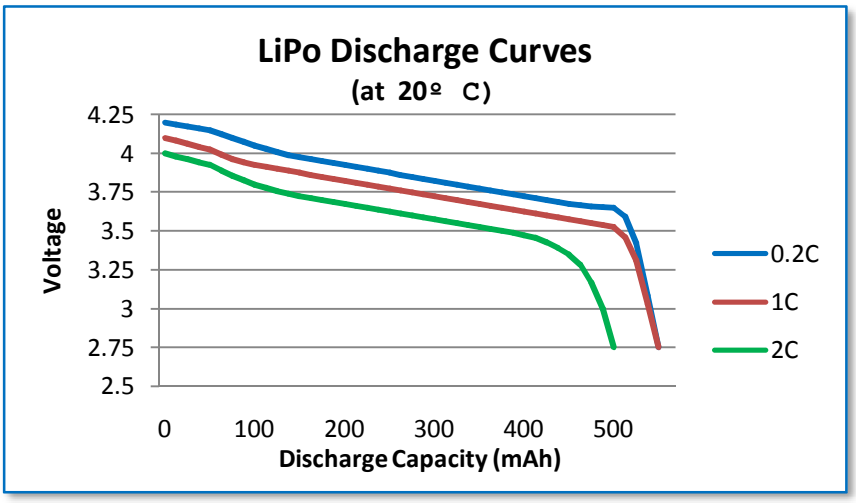
- Nominal voltage – My advice to you is forget you ever heard the nominal voltage spec for the battery you are working with. Nominal voltages generally reflect a certain point in the battery's discharge curve under conditions your battery may never see in the real world. For example, a LiPo cell has a nominal voltage of 3.6V but its operating voltage range is from 4.2V to 3.0V. As you

can see in the discharge curves below for a LiPo battery, under a light load (the blue line) the curve flattens out around 3.6V just before falling off the cliff. Other than this short period of time, it would be hard to correlate anything on these discharge curves to 3.6V. The two critical voltages you need to be concerned with are maximum voltage for charging and the rated cut-off voltage for discharging. To get the full life of your batteries, these two voltages should never be exceeded.

- Factors effecting output - Load and temperature are the primary factors that create the discharge curves for rechargeable batteries. In the case of load, the variable resistor in our battery model acts much like any other resistor and the output decreases as the load increases. Temperature appears to make this variable resistor act the opposite of a real resistor, as the temperature decreases the apparent resistance increases, reducing the voltage at the output. The combination of a heavy load and low temperatures can result in a considerable decrease in the output voltage. As shown in the graph below, the output of a LiPo battery can decrease by over a volt from light load at high temperature to a heavy load at low temperature. For non-rechargeable batteries, the age of the battery also impacts the voltage. As these batteries get older (whether they are in use or not), the apparent resistance increases causing the output to decrease.



- Discharge curves – All battery technologies have a characteristic discharge curve, typically plotting voltage against mAh capacity (or age). While useful for comparing battery chemistries, once you select a battery technology you need to look at the discharge curves for specific battery models. Variations on the chemistry and battery construction yield different discharge curves. Specifically, you need to pay attention to the curves for the load you expect to place on the battery and the temperatures you expect your product to encounter. These two factors can have a significant impact on the charge duration your batteries will have in actual use. The discharge curves in the graph below are for a LiPo battery at three different loads (rated discharge rate multiplied by 2, 1 and 0.2). You really need to look at both the voltage at temperature and load graph and the discharge curves to get a feel for how a battery will perform in your application.



- Rated current – Batteries typically have both maximum continuous current and maximum pulse current specs. These values should never be exceeded. Best case you can severely decrease the life of the battery. Worst case, with lithium based batteries the result can literally be smoke and flames. If your design requires getting close to these rated currents, contact the battery manufacturer to discuss your application to make sure your product will be safe and operate properly. You may also consider using two batteries in parallel to increase the available current (this carries its own set of issues that are beyond the scope of this discussion).
- Charging – A thorough discussion of battery charging would take a book several times the size of this one but there are a few basics you should understand about battery charging. The first thing you should be aware of is each battery technology has certain charging algorithms that are optimized for that technology. Using an algorithm designed for a different technology can have results ranging from poor charging to disastrous (back to smoke and flames again). The basic parameters of battery charging are voltage, current and time. In general, the higher the voltage and/or current the shorter the charging time and each battery technology will have limits on the maximum voltage and current. Some types of batteries respond well to charging with high current pulses while others won't. Some will benefit from a trickle top-off charge while others won't. Fortunately, a number of semiconductor companies have battery charger chips for specific battery chemistries so you don't need to get bogged down in this particular detail.
- Charge duration vs battery life – It is somewhat important to make a distinction between charge duration and battery life. Charge duration refers to the length of time from charging until the output reaches the cut-off voltage. Charge duration can be impacted by improper or incomplete charging, load, temperature and battery age. All batteries have a finite useful life (based on number of charge cycles or age) and battery life literally refers to the useful life of a battery. Battery life can be impacted by aggressive charging and over charging, exceeding the maximum current rates and using the battery below its rated cut-off voltage. Long charge duration and long battery life are not mutually exclusive but to achieve both you must treat a battery gently. Deep discharging to increase charge duration will reduce the life of a battery (as will very aggressive charging algorithms). Particularly in the case of LiPo batteries, the discharge curve is so steep when the cut-off voltage is reached there is minimal benefit and a high penalty in battery life

reduction for going below the cut-off voltage. The term “battery life” is often used when referring to charge duration but when discussing ways to save power, what you do to increase charge duration will generally lead to longer battery life too.

- Self-discharge – All battery chemistries suffer some amount of self-discharge over time. This generally isn’t a concern as far as your product design is concerned. However, electronics have reached such low current levels that the battery’s self discharge may play a significant factor in your product’s charge duration and may even be higher than your product’s discharge rate.
- Multiple batteries – Batteries are easily put in series to increase voltage, in parallel to increase current capability or both. This can be done with little concern with non-rechargeable batteries. For rechargeable batteries, there are a number of considerations primarily around charging. The details on this are also outside the scope of this book but if you use multiple batteries in your product, you should use a battery pack instead of individual cells to ensure the cells are from the same manufacturer and roughly the same age
- Physical considerations – There are a few physical considerations you must take into account in your product design when using certain types of batteries. For example, some types of batteries may out-gas during charging (usually a sign the battery is being abused) and require venting to the outside world to avoid explosion. The lithium based rechargeable batteries can literally swell during charging and require some amount of room to expand. Battery manufacturers usually provide information if special considerations are needed.

For the most part, your application will dictate the type of battery you use. In some cases you may have a choice of battery types or may have to decide between battery types when more than one would be suitable for you application. On the next page is a high level overview of the pro’s and con’s of the most popular battery types.

If you are new to designing battery powered products, it will be well worth your time to do some in research into the various battery technologies Even if you are an old hand at designing with batteries, if you are considering a battery technology you haven’t used before for you product you should do some in depth research in that technology to understand it’s intricacies. Manufacturers of batteries intended for industrial applications or OEM use generally provide considerably more details in their spec sheets than those for consumer oriented batteries. These spec sheets and app notes are a good source of valuable information.

Battery Type	Pro's	Con's
Lithium coin cell	Small Inexpensive Readily available Low self discharge rate Steady voltage may power circuits directly	Not rechargeable Requires battery door in product Limited capacity Don't tolerate reverse current well May be subject to shipping restrictions
Standard cell (AAA, AA, C, 9V, etc.)	Small Inexpensive Readily available Low self discharge rate	Not rechargeable Requires battery door in product 1.5V cell voltage means multiple cells - or - 1.5V cell voltage means step-up regulator
Lead-acid (traditional, sealed and gel types)	Extremely high energy density Fairly tolerant of charging abuse Standard sizes readily available Inexpensive Voltages match well with solar panels	Environmentally nasty Usually large and heavy High self discharge rate
Nickel metal Hydride (NmH)	Good energy density / weight ratio Tolerant of charge/discharge abuse Available in wide array of sizes Low self discharge rate	"Memory" effect
Lithium ion & Lithium polymer	Good energy density / weight ratio Available in wide array of sizes No "memory" effect Low cost Low self discharge rate	Charging abuse can lead to fire Discharging abuse can lead to fire May be subject to shipping restrictions May require battery door in product

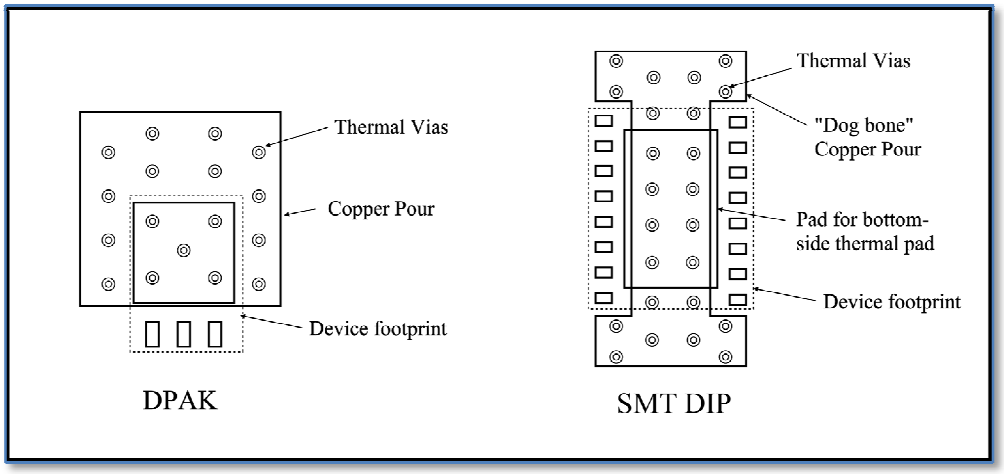
Keep it cool

As semiconductor geometries have shrunk, in recent years leakage current has become a significant component of the overall power consumed by ICs. As parts heat up, their leakage current typically increases. It is not uncommon for parts to consume twice as much current at their highest rate temperature than at their lowest. For example, the AD8226 op-amp is rated for -40°C to 125°C. The quiescent current ranges from 325uA at -40°C to 425uA at 25°C to 600uA at 125°C. This is nearly a 100% increase across the temperature span and nearly a 50% increase from "room temperature" to the maximum temperature. You should conduct your current measurements at the temperature your product will normally operate at if not at the temperature extremes too.

Controlling Heat

Even if you don't have the luxury of airflow in your product, there are a number of things you can do to keep the heat under control and reduce the impact of heat on your power consumption:

- If your product is vertically mounted, place as much circuitry as you can below the main heat generating parts.
- If you have the option for voltage regulators and other heat generating parts, select packages with bottom side ground or thermal pads. Dissipating heat into the circuit board can help localize the heat buildup and maintain a lower air temperature within the enclosure.
- For products that operate in high temperatures and have more than a few ICs, select part packages based on what you want to do thermally for a part. For parts that drive large loads or use high clock speeds it can be beneficial to select a package with a low thermal resistance to help get the heat away from the die. On the other hand, if you must place other parts near heat sources on your board you can choose a package with a higher thermal resistance for those other parts to reduce the heat transferred from the PCB to the die. Most surface mount ICs have several options for package styles that can have a wide range of thermal resistances. For instance, Texas Instruments offers the 74LV74 in 6 different packages with thermal impedances ranging from 47°C/W to 127°C/W.
- To fully realize the heat dissipating potential of low thermal coefficient packages with large tabs or bottom side thermal pads, you have to place several thermal vias in the pad to tie this pad to the internal ground plane or a large copper pour on the back of the PCB. You also need to pay special attention to the datasheet on parts with an exposed bottom side pad. These exposed pads are often connected to ground internally but not always and on some parts the pad may be the only ground connection for the part. On other parts, the pad may not be electrically connected and can be safely grounded or it may be the negative voltage supply on dual supply analog parts (which may or may not be ground in your design). The datasheets usually contain details on the copper area and other PCB layout requirements to achieve the specified thermal resistances. The diagram below shows a typical arrangement for a DPAK and SMT DIP packages with a thermal copper pour and vias to connect to an internal ground plane or back-side copper pour. Most of the IC manufacturers that have packages with bottom side thermal pads provide app notes or even on-line calculators to help determine the minimum size of the copper area and number of thermal vias you need for a given package.



- With thermal vias, more isn't always better since they can also disrupt the spread of heat across a copper pour. Some assembly houses may complain about thermal vias in device pads robbing solder from the pad. If so, reducing the via size or covering the via on the opposite side of the board with solder mask can help minimize the amount of solder that may seep into the via holes.
- When using the PCB for heat sinking, keep in mind that FR4 and other laminates that PCBs are commonly made with are very poor thermal conductors. You are really spreading the heat through the copper in/on the PCB instead of transferring the heat into the PCB. It is best to use at least 2 oz copper for the outer layers and 1oz copper for inner layers. The heavier copper isn't significantly more expensive than the "standard" 1 oz and ½ oz copper used for most PCBs but does provide significantly better heat transfer across the ground plane and copper pours.
- If you have traces on your board that carry more than a few amps, the traces themselves can be a significant source of heat if you aren't careful. There are a number of good on-line PCB trace width calculators you can use to help prevent this. The maximum allowed temperature increase for the trace is an input for most of these calculators. If a trace normally carries high current you should set this to 5°C or 10°C. If the high currents are of a fairly low frequency and short duration, you can usually go as high as 25°C max temperature rise (but first make certain your product isn't subject to any specifications that restrict the max temperature rise of traces). The overall trace resistance is also a function of trace length so you can reduce this resistance and resulting generated heat by shortening the trace length. Vias can have significantly higher resistance than copper traces causing them to generate additional heat so above a few amps you should use multiple vias instead of a single via.
- Simply increasing the space between heat generating parts and the micro can drastically reduce the amount of heat the micro is exposed to.
- If possible, heat sink your heat generating parts to the enclosure. You may be able to do this directly but also indirectly be placing your heat generating parts near mounting holes on the board so the mounting hardware can help carry the heat out of the circuit board.
- Pay the premium for more efficient voltage regulators. The power that an inefficient regulator wastes is generally turned into heat which can increase the power consumed. You aren't likely to get into a thermal runaway condition but to some degree this increased power consumption

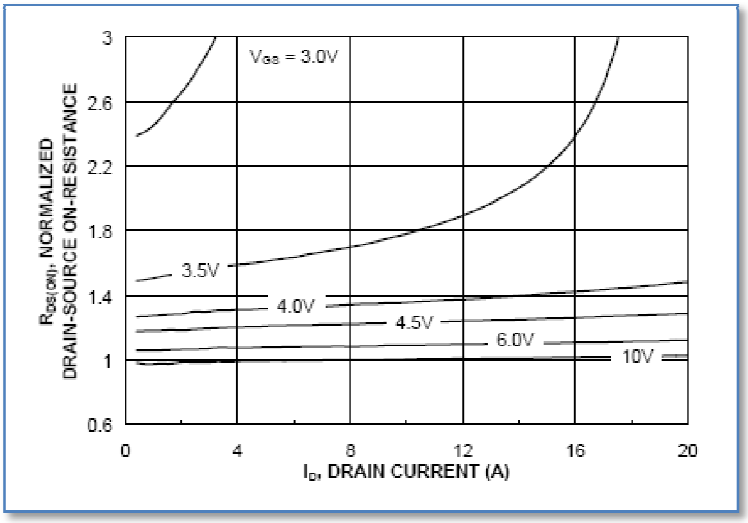
generates more heat which increases power consumption which generates more heat …. you get the point.

Driving loads

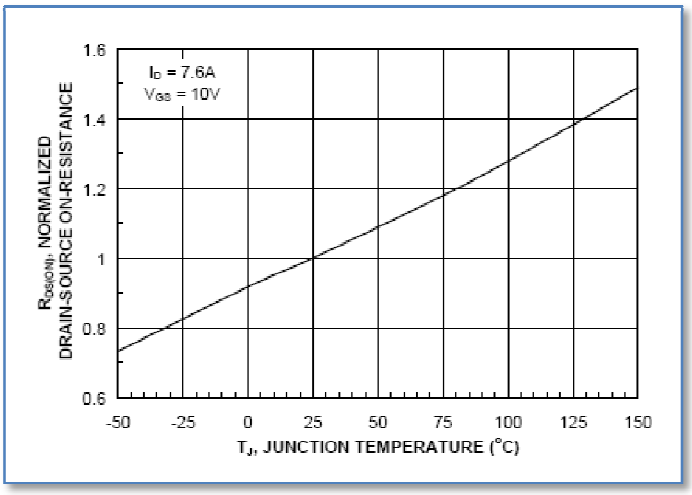
You may not expect to read about multiple amp loads in a book titled "Low Power Design". Perhaps it reflects on a poor choice for a title or it could be an indication of how wide-spread the push for energy efficient products has become. The gigahertz micros in laptop computers can draw tens of amps when operating at full speed. In many embedded applications it is quite common for a micro drawing a few milliamps to control motors or solenoids that require many amps of current. Even if these current ranges are way above what your design has to deal with, the information presented here may be applicable to your design.

When driving a large load such as a DC motor, solenoid or a cluster of LEDs, there are a number of things you can do to ensure you drive them efficiently to help reduce heat buildup:

- You already selected the lowest RDSon FET you could find but are you taking full advantage of its capabilities? Make sure your gate voltage is such that the FET is fully turned on and operating in its lowest RDSon range. Most N-channel FETs usually need a gate voltage in the 8-10V range to fully turn on so simply driving the gate with a GPIO won't put the FET in its lowest RDSon range. Logic level gate FETs may be better in this regard but typically they just have a lower minimum turn-on threshold and may still need over 5V to fully turn on. In these cases you should consider using a P-channel FET or even a gate driver IC to drive the gate of the N-channel FET with the voltage it is switching (or use a step-up regulator or voltage doubler circuit to provide a higher voltage if the input voltage exceeds the maximum gate voltage). The graph below shows the impact of gate voltage on Rds(on) for the Fairchild FDS8449 N-channel FET. The FDS8449 has a max gate turn-on threshold of 3V but as you can see at 3V the Rds(on) is 2.4X higher than at 10V at no load, much higher as the load increases.



- When using a P-channel FET to drive a load, a GPIO may not drive the gate high enough to completely turn off the FET so you may be leaking power through the FET. This can often go un-noticed since the amount of power is too low to activate the load.
- A P-channel FET of similar rated voltage and current as an N-channel FET will typically have 50-100% higher Rds(on) than the N-channel FET. With Rds(on) specs on modern FETs in the double-digit milliohm range even doubling the Rds(on) produces a fairly low value. However, that is simply wasted power that can easily be eliminated if low-side switching is an option for your application. This is also important to keep in mind if for some reason you can't address one of the issues discussed here that prevents the FET from operating close to it's lowest Rds(on), changing the type of FET may alleviate the problem. If you have a P-channel FET operating at 2X its lowest Rds(on) then you could possibly reduce the Rds(on) by a factor of 4X by using an N-channel FET.
- Just like a resistor, the Rds(on) of a FET increases with temperature. The graph below shows the impact of temperature on Rds(on) for the Fairchild FDS8449 N-channel FET. As you can see, a 50°C increase in temperature results in a nearly 20% increase in Rds(on). Even if your product is normally used in a room temperature environment, a 20-50°C temperature rise at the FET's die isn't uncommon (another reason to operate the FET in its lowest Rds(on) range). This is another situation where keeping a part cooler helps prevent wasting power and similar to the earlier discussion on voltage regulator efficiency, as the Rds(on) increases the part will get hotter, increasing the Rds(on) and so on. A hot FET and a non-optimal gate voltage can combine to generate a lot of excess heat and waste a lot of power.



- Similar to LEDs, electromechanical devices can often be driven with a PWM to reduce power without impacting the performance of the device. Solenoids can often be "kicked" with a several hundred millisecond pulse to actuate them and then driven at as low as 30 or 40% duty cycle to keep them actuated. Depending on size, a DC motor may require a "kick" for up to a few seconds to get it up to speed and more than a 50% duty cycle to avoid slowing down but with such large loads, even a 10-20% savings can be a considerable reduction in power consumption.

If your application requires mixing ultra-low power design with high currents or other potential heat sources, consider buying or renting an IR thermal camera. The camera will help you find hot spots on

your board and access the effectiveness of your heat spreading/isolation efforts. Don't be tempted to skimp and use a laser pointer IR thermometer instead of the camera. The IR thermometer has a fairly narrow range of view and only displays the temperature where you point it. An IR thermometer may not accurately measure the temperature of a small hot spot like one caused by high current flowing through a single via. The beauty and value of the camera is it can show you even tiny hotspots where you don't think about looking for them. The first time I used a thermal camera it did just that, allowing us to fix an issue at the prototype stage that would have likely lead to field failures and high warranty costs. That camera more than paid for itself in a matter of hours.

Firmware

As mentioned in the introduction, a product's hardware design will establish the minimum level of power consumption for the product. For many types of products, the firmware will determine the highest level of power consumption. More importantly, the firmware will also determine how much time is spent at the minimum level of power consumption.

Some power saving concepts will be presented that can be applied whether your firmware is in assembly language or C. There are also a number of techniques that can be applied when using C to force more power efficient code than a compiler would normally generate. Some of these techniques will go against the basic principles of structured code. Highly structured code unfortunately is also power-wise very inefficient because of all the instructions the micro has to execute because of the structure that don't directly contribute to completing the task at hand.

Power saving in firmware is all about (1) eliminating unnecessary clock cycles the micro uses while performing a task and (2) putting the micro in a low-power state as often and for as long as possible for the given application. To a large extent, eliminating unnecessary clock cycles naturally leads to spending more time in a low-power state but the time spent in a low-power state is more heavily influenced by the code structure. Keep in mind that everything your code does that it doesn't need to do or doesn't need to do as frequently as it does is just wasting power. Always keep in mind the equation for current used during a specific event:

$$I_{event} = (Time_{operating} \times I_{operating}) + (Time_{sleep} \times I_{sleep})$$

Choice of programming languages

For the ultimate power savings you must be in control of the instructions your micro executes as much as possible. Carefully written assembly language can provide lower power consumption than the best compiled code but is slower to develop and harder to maintain. If you choose to use a compiled language, C is probably the best choice since you can achieve a decent level of control over the compiled code and there won't be as much run-time code generated by the compiler that you aren't aware of as there can be with C++. A considerable amount has been written about the suitability of C++ for embedded programming. Whatever your stance on this, it is hard to debate that well written C code is more efficient than equally well written C++ code.

Main function vs the rest of the time

Very few products actively perform their main function 100% of the time yet that is where many engineers spend their timing trying to improve efficiency. What the firmware does while inactive often has a bigger impact on power consumption than what it does while active. Maximizing the time the micro spends in the highest power saving mode suitable for the application is key to reducing long term power consumption.

Frequency of events

Make sure your product only performs its main function as often as is really necessary. Except for low level control systems, it is fairly rare that monitoring physical conditions needs to be done multiple times per second or even per minute. For example, if monitoring temperature is a primary activity you may need to do you may be able to get by only checking the temperature once every few minutes. The same can be said for monitoring battery voltage. Even when response time is critical, for most physical conditions a longer interval can be used when the condition being monitored is well within normal bounds and then the interval reduced as the condition being monitored starts approaching a critical threshold.

Polled vs interrupt driven events

Particularly with slow peripherals such as UARTs, interrupt driven firmware will use considerably less power than pollIng firmware. A micro may execute thousands of instructions during the time it takes for a UART to transmit or receive a byte of data. Even at a relatively fast 115.2K baud, an 8Mhz processor will burn almost 700 clocks in one byte time, drop that to 19.2K baud and it goes up to over 4,100 clocks per byte. On the other hand, with a fast peripheral such as an A/D converter or SPI/I2C controller the interrupt processing overhead for a slow 8-bit micro may use more power than a polling loop.

DMA vs firmware loops

If an application requires moving large amounts of data to or from a peripheral (or even small amounts of data with slow peripherals), it is generally more power efficient to move the data with a DMA controller than in a firmware loop. This is an area where you must do your homework up front to ensure you select a micro that supports DMA operations with the required peripherals and that the DMA controller is operational with the micro in a sleep mode.

UART FIFOs

If DMA is not an option and your micro has transmit/receive FIFOs in its UARTs, take advantage of these FIFO so that your firmware doesn't take an interrupt for each byte transferred. For instance, when transmitting data you can fill the FIFO and only take an interrupt when the last byte in the FIFO has started shifting out. Many newer micros with UART FIFOs allow you to set the point where the interrupt is generated so you don't have to wait for the last byte to get an interrupt if you have over-run/under-run concerns.

Structured code

Highly structured source code is nice to work with but unfortunately can run considerably slower and consume considerably more power than less structured code. You don't have to forget everything you learned about structured code but violating some principals of structured code can help save power:

- Global and fixed address variables – Global variables can save a considerable amount of power by not passing parameters to functions (particularly for older 8-bit micros or when using external

RAM). Fixed address variables can help reduce the power used when calculating addresses for structure members. If you have the good fortune of having extra RAM, using fixed address variables in functions can be much more power efficient than working with temporary variables that are addressed relative to the stack pointer.

- Put small functions in-line – If you have small functions that are called frequently you will save power by placing the code for those functions in-line. Particularly with older 8-bit micros, the overhead for calling and returning from a function can take dozens of clock cycles. Some compilers support using function calls in your source code to help keep the source code cleaner but will place the code for the function in-line instead of doing actual function calls.
- Peripheral "drivers" – If you use a micro with multiple instances of a peripheral type, say 4 UARTs for example, you can save power by having a set of code for each UART instead of generic code that is passed a parameter to specify which UART to use. This avoids the parameter passing and allows you to hardwire the peripheral register addresses in the source code instead of calculating addresses at run time. Particularly with 8-bit micros this can yield a considerable savings for frequently used peripherals. These types of functions are typically fairly small so you won't necessarily use a lot of program space by doing this.

Value added and non-value added code

Lean manufacturing has a concept of value added steps and non-value added steps. Value added steps directly contribute to producing the end product. Non-value added steps are essentially overhead steps that may be necessary but don't directly contribute to producing the end product. An example of a non-value added step would be moving sub-assemblies from one section of the manufacturing floor to an assembly line. It is natural to focus on the value-added code for power optimizations because that is where the primary tasks for your product are. For the ultimate power savings, you should analyze your code to identify the non-value added sections of code that don't get much consideration. You need to determine if the non-value added sections of code are absolutely necessary, "nice to have" or not really necessary. For the non-value added sections of code that can't be eliminated, can they be streamlined, executed less frequently or grouped with other non-value added functions (particularly if the micro is being taken out of a low-power state to execute the non-value added code) in order to save power?

The power of loops

Firmware loops can be very effective in reducing code size and making the source code cleaner and more readable. They can also be huge wasters of power simply because of the overhead instructions required to implement the loop. Managing the loop counter, checking for the loop termination and the jump back to the start of the loop can all be considered non-value added code. Unless you are severely constrained on code space, unrolling frequently executed loops into a repetitive series of instructions can save a considerable amount of power. This is most easily done for loops with a small, fixed number of iterations. Even loops with a variable number of iterations can be made more power efficient this way. The test for "loop" termination would need to be done between each series of instructions but the jump would only be taken once instead of each time through the actual loop. This

can be a considerable power savings particularly for micros performing instruction pre-fetches that are discarded at the end of each pass through the loop.

Know your variables

This may seem like a no-brainer but particularly with C compilers what you see in the code isn't always what you get in execution. Pay special attention to this with variables used in loops since that one line in your source code can be executed hundreds or thousands of times.

- Variable size – Particularly with 8-bit micros, make sure your variable sizes aren't larger than they need to be. As shown in the code segment below, simply incrementing a 32-bit value can turn into 3 tests for overflow, 3 add instructions and as many as 8 memory accesses. In most cases this is just a waste of code space but if the variable is a free running counter it will be wasting power every 256th time it is incremented. A 32-bit add or compare is even worse since all four bytes of the variable must be operated on every time. Most compilers will size an "int" to be the word size of the micro but to be sure you should explicitly declare variables as int8, int16, etc. (or whatever syntax your compiler uses).

```
If "var" is a 32-bit value, on an 8-bit micro the C statement "Var++" turns
into this sequence of assembly language instructions:

      LOAD  R1,@var_0              ; byte 0
      INCR  R1
      STORE @var_0, R1
      TST   OVERFLOW
      BRZ   incr_done
      LOAD  R1,@var_1              ; byte 1
      INCR  R1
      STORE @var_1, R1
      TST   OVERFLOW
      BRZ   incr_done
      LOAD  R1,@var_2              ; byte 2
      INCR  R1
      STORE @var_2, R1
      TST   OVERFLOW
      BRZ   incr_done
      LOAD  R1,@var_3              ; byte 3
      INCR  R1
      STORE @var_3, R1
incr_done:
```

- Signed vs unsigned variables – Arithmetic operations on signed and unsigned values aren't handled the same way by compilers. To avoid extra processing (and weird results from the math operations) be careful about the declarations and similar to the size, explicitly declare variables signed or unsigned and don't assume an "int" is one or the other. Also be very careful about mixing signed and unsigned values in math operations.
- Variable alignment – Some 32-bit micros require 16-bit values to be on 2-byte boundaries and 32-bit values to be on 4-byte boundaries and will generate exceptions or bus faults for unaligned accesses. More sophisticated micros can handle mis-alignments and will happily turn an unaligned 32-bit access into two, three or four memory accesses without you knowing it. C compilers typically have options to allow misalignments or force certain alignments. No so much related to power usage, compiler forced alignments can be very wasteful of RAM space by placing every

variable on a 4-byte or even 8-byte address boundary. This can be particularly wasteful with large structures with various sized elements and arrays of 8 or 16-bit elements. The "pragma pack" directive can help you in this regard with structures but should be used with care since it puts you in charge of determining the alignment.

- ASCII vs Unicode – If your product interfaces with text strings to a Windows application you may need to support 16-bit Unicode values, otherwise 8-bit ASCII should be adequate. If you are stuck with Unicode and your product requires a proprietary driver or uses a Windows app, you should be able to save power by using ASCII in the firmware and translating to Unicode in the PC based code.

Timers

- Use the largest clock pre-scaler that provides the resolution your firmware needs. The pre-scaler is typically a 4 to 8 bit counter while the counter/timer may be 8, 16 or 32 bits. Letting the pre-scaler run faster so the timer runs slower can reduce power considerably, particularly for free-running timers.
- Software based timers running on a tick interrupt are easy to implement and may be necessary if your application requires more than a few timers to be running simultaneously. Dedicated timers may be more power efficient IF the time-out period can be achieved with a single terminal count interrupt from the timer.
- When using a periodic tick interrupt for software timers, use the longest tick interval your code can tolerate. If sections of your code require tight timing it will usually be more efficient to use one timer with a longer tick interval for general use and another timer for with a short tick interval for the tight timing.
- For a timeout protection timer that doesn't have tight timing requirements, consider using an RTC alarm interrupt for a 1 second (or longer) timeout. The RTC running at 32Khz should be much lower power than an 8 or 16 bit timer using a clock divided down from the micro's much faster clock. This also provides a means for long timeout periods without taking periodic tick interrupts.
- Turn off timers when they aren't being used. For dedicated timers, this is usually just a matter of selecting the right mode for the timer so it stops automatically when it reaches the terminal count. If software based timers are appropriate for your application, turn off the free-running timer when it is being used.
- For ultra-low power, when possible use a timer that counts down to 0 or counts up to all ones and generates an interrupt. Using a counter and match register containing the terminal count value requires more circuitry to be active and will consume more power.

Compilers

If you are writing firmware in any high level language you need to become intimately familiar with the compiler and learn what it does well and what it doesn't it. The only way to do that is to write some code and then examine the assembly code it generates.

- Efficiency – Every instruction your micro executes that isn't required is wasted power. Compiler efficiency is usually a case of you get what you pay for. Free and low-cost compilers based on the

GNU compiler typically produce code 2X to 5X larger, slower and less power efficient than a compiler written for a specific architecture.

- In-line assembly code – If you decide to use assembly language for time critical sections of code or for other reasons, you need to research how your compiler handles in-line assembly instructions. In some compilers, the registers you think you are using are actually memory based variables. If your assembly code uses many variables or contains a loop that is executed more than a few times, you are generally better off calling a function written in actual assembly code since the function calling overhead uses less power than the pseudo assembly code. If you do use in-line assembly language, review the compiler generate assembly language before and after your assembly language code to see how much overhead the compiler imposes for saving/restoring the micro state information.
- Compiler options – Compilers usually have many options, some of which can greatly increase or decrease the power efficiency of your code. A few things to look for:
    - Position independent code – Disable this option unless you absolutely need it. The relative addressing required for position independent code will use more power than fixed location code on every jump/call instruction executed.
    - Optimization options – Compilers typically provide options to optimize the generated code for speed or for size. The optimizations made for speed should also improve power efficiency since fewer clocks uses less power but will result in larger programs. If you are tight on code space, check to see if your compiler supports optimization on a per file basis so you can optimize the most frequently executed sections of code.

Structures

Structures are great for organizing variables but you need to consider whether this convenience is worth the cost in power compared to individual variables. A few things to consider:

- Every time a structure element is used the micro has to add the element offset to the structure base address. On most 32-bit micros this is achieved with an indexed addressing mode so no additional clock cycles are required. On a low-end 8-bit micro this requires code to calculate the address, taking 8 to 10 instructions or more.
- Arrays of structures further complicate the math involved in calculating addresses. To calculate the offset into the array, the array index must be multiplied by the structure size and that is done in software on most micros used in embedded applications. If the array only contains a few instances of the structure it can be considerably more power efficient to have individual structures with another variable containing a pointer to the structure to use. Another technique for use with larger arrays will be discussed in the “Arrays and structures” section.
- Don’t assume your compiler is smart enough to calculate the base address for a particular structure in an array once and use it for several consecutive lines of C code that access that structure. There is a good chance it will calculate that base address for each line of C code. To use less power, use a pointer to the structure in this situation, the compiler should only calculate the address based on the offset for structure members.

Don't do this:

```
SOME_STRUCT       struct_array[8];

struct_array[array_offset].mem1 = 0;
struct_array[array_offset].mem2 = 0x55;
struct_array[array_offset].mem3 = 0xaa;
```

Do this instead:

```
SOME_STRUCT       struct_array[8];
SOME_STRUCT       *ssa_ptr;

ssa_ptr = &struct_array[array_offset];
ssa_ptr->mem1 = 0;
ssa_ptr->mem2 = 0x55;
ssa_ptr->mem3 = 0xaa;
```

- One place where structures may work to save power is in parameter passing. Particularly with low-end 8 bit micros, passing multiple parameters can be considerably more expensive in terms of power and time than passing a pointer to a structure containing those parameters. The example below illustrates this, even using the structure for the return value.

Don't do this:

```
ret_val = some_func(param1, param2, param3, param4);
```

Do this instead:

```
typedef struct{
      uint8       param1;
      uint8       param2;
      uint16            param3;
      uint16            param3;
      uint8       ret_val;
} SOME_FUNC_PARAMS;
SOME_FUNC_PARAMS  sf_params;

some_func(&sf_params);
```

Switch statements

It can be easy to fall into the trap of thinking in a switch statement the micro examines a variable and auto-magically goes to the correct block of code for that value of the variable. Since they are really a sequence of if-then-else statements, frequently executed switch statements can be very wasteful of power. Here are several ways to make them more efficient, some of them can be combined for even greater efficiency improvements:

- Arrange the order of the case statements so that the most frequently executed cases are listed first. You should check the compiled code to make sure the assembly code checks for the cases in the order they are listed or abandon the switch statement and implement your own sequence of if-then-else statements to ensure the order you want.
- You may be able to sacrifice some simplicity in the code and do a binary decode on the switch variable. This can reduce an eight case switch statement from as many as seven tests to a series of three tests as shown below. If a few values are encountered significantly more often or are more time critical than the others, you can specifically test for those values before starting the

binary decode. If you do this, remove the code for those values in the decoder code for clarity and to reduce code size.

Don't do this:

```
switch(value)
{
      case 0:
            ....
            break;
      case 1:
            ....
            break;
            .
            .
            .
      case 7:
            ....
            break;
}
```

Do this instead:

```
if ( value & 0x04 )
{     /* bit 2 is set, value is 4 to 7 */
      if ( value & 0x02 )
      {     /* bit 1 is set, value is 6 or 7 */
            if ( value & 0x01 )
            {     /* bit 0 is set, value is 7 */
                  ....
            }
            else
            {     /* bit 0 is clear, value is 6 */
                  .....
            }
      }
      else
      {     /* bit 1 is clear, value is 4 or 5 */
            if ( value & 0x01 )
            {     /* bit 0 is set, value is 5 */
                  ....
            }
            else
            {     /* bit 0 is clear, value is 4 */
                  ....
            }
      }
}
else
{     /* bit 2 is clear, value is 0 to 3 */
      if ( value & 0x02 )
      {     /* bit 1 is set, value is 2 or 3 */
            if ( value & 0x01 )
            {     /* bit 0 is set, value is 3 */
                  ....
            }
            else
            {     /* bit 0 is clear, value is 2 */
                  .....
            }
      }
      else
      {     /* bit 1 is clear, value is 0 or 1 */
```



```
                        if ( value & 0x01 )
                        {     /* bit 0 is set, value is 1 */
                              ....
                        }
                        else
                        {     /* bit 0 is clear, value is 0 */
                              ....
                        }
                  }
            }
      }
```

- Using several if-then-else sequences testing for ranges of values of the switch variable with each test having its own if-then-else sequence can considerably improve the efficiency. The example below shows splitting what would be a 16 value switch statement into two "if" statements each with an eight value switch statement, reducing the worst case from 15 tests to 8 tests. Breaking it down further to four "if" statements each with a four value switch statement reduces the worst case to six tests.

```
if ( ( value > 0 ) && ( value << 8 ) )
{
      switch (value)
      {
            case 0:
                  ....
                  break;
                  .
                  .
                  .
            case 7:
                  ....
                  break;
      }
}
else if ( ( value > 9 ) && ( value << 16 ) )
{
      switch (value)
      {
            case 9:
                  ....
                  break;
                  .
                  .
                  .
            case 15:
                  ....
                  break;
      }
}
```

- Nesting switch statements can produce similar improvements in the worst case number of tests required. To do this effectively you really need two variables or a variable that can be cleanly split into two fields like a "mode" for the first level switch and "command code" for the second level switch statements. The example below shows how an instruction opcode parser could be done with the opcode split into an instruction type field and command code field.

```
switch (opcode & TYPE_FIELD)
```



```
{
      case (MATH_INST)
            switch (opcode & CMD_FIELD)
            {
                  case (ADD_INST)
                        ....
                        break;

                  case (SUBTRACT_INST)
                        ....
                        break;

case (MULTIPLY_INST)
                        ....
                        break;
            }
            break;

      case (LOGIC_INST)
            ....
            break;

      case (BRANCH_INST)
            ....
            break;

}
```

Arrays and pointers

- If you use arrays of structures, you can also use arrays of pointers to the structures in the array to save power. The structure addresses will be pre-calculated by the compiler and in the example below the array of pointers will be initialized at reset. The math needed to calculate the offset into the array of pointers at run time will likely be a shift and add instead of the multiply and add generally required to calculate the offset into the array of structures, greatly reducing the power used every time a pointer to one of the structures in the array is needed. Using the appropriate compiler directives, the array of pointers could be made constants and stored in program memory if your RAM space is limited. Be careful doing this with Harvard architecture micros since loading data values from the program memory space can turn into a sequence of several instructions instead of a simple memory access.

```
SOME_STRUCT       struct_array[16];
SOME_STRUCT       *ssa_ptrs[16] = {
                        &struct_array[0],
                        &struct_array[1],
                        &struct_array[2],
                              .
                              .
                              .
                        &struct_array[15] };
```

- For ultimate power efficiency, change a case statement into a “jump table” using an array of pointers to functions using the switch variable as the array index. Even with a sanity check to ensure the variable is a valid index into the array this can be much more power efficient and quicker than other methods. Even in an application like command parsing where there can be

dozens of valid values and many unused values, the large array this produces can still take much less program space than the sequence of if-then-else statements the switch statement would produce. The example below shows such an array of function pointers and includes a pointer to an invalid function handler to deal with unused values within the range of valid values.

```
void  *jump_table[TABLE_SIZE] = {
                                (void *)&func_1,
                                (void *)&func_2,
                                (void *)&func_3,
                                      .
                                      .
                                (void *)&invalid_func,
                                      .
                                      .
                                (void *)&func_n };

if ( selection < TABLE_SIZE )
{
      (*jump_table[selection])(selection, param2);
}
else
{
      invalid_func(selection, 0);
}
```

Floating point math and complex calculations

Even a small 8-bit micro is capable of performing incredibly complex floating point calculations. However, doing so can be incredibly expensive in terms of time and power, with simple floating point operations taking well over a thousand clock cycles. The table below lists the number of clock cycles required for various floating point operations on an 8051 with a popular C compiler (from Silicon Laboratories white paper "CIP-51 Performance for Standard Library Math Routines" Rev. 2.1 12/03).

Floating Point Operation	Clock Cycles
Addition	1,284
Subtraction	1,356
Multiplication	1,368
Division	8,244
Square root	23,232
Sin	35,136
ArcSin	83,892

Here are several things to consider and a few tips for saving power if your application involves complex calculations

- The first thing to consider is whether your application absolutely, positively REQUIRES floating point math. There is a common misconception that floating point math is more accurate than scaled integer math. In reality, floating point math is very good at expressing extremely large or small numbers but depending on the floating point library you use it may not be as accurate as scaled integer math. A colleague of mine once ran an experiment where his code executed the

code segment below. Much to our surprise, after completion of the loop "result" was not equal to 1.0 as would be expected, it was within several decimal places but was not 1.0.

```
result = 1.0;
for ( x = 0; x < 10000; x++ )
{
      result = result * 1.0;
}
```

If floating point math isn't a hard and fast requirement for your product for some reason, after your floating point firmware is working, try a little experiment. Take some current measurements and calculate the power used during your product's operation where the floating point math takes place. Next, replace the floating point math with scaled integer math and recalculate the power used. Then you can decide if what you think you gain with the floating point math is worth the expense in power consumption.

- In many cases, a value can be pre-calculated instead of calculated in real time. Consider a theoretical application that needs to calculate a force vector based on data from a 3-axis accelerometer. The exact force is not important, the application only needs to know when a certain threshold has been exceeded in order to set an alarm. The equation for the force vector is

$$force = \sqrt{x^2 + y^2 + z^2}$$

  where *x*, *y* and *z* are the g force values for the 3 axes. In this case, the square of the threshold force scan be pre-calculated and the sum of x^2, y^2 and z^2 compared to that value instead of performing the square root calculation.
- In some cases, input values and partial results may be retained from previous calculations so that a calculation for a partial result need not be performed if the input value hasn't changed. In the previous example with the accelerometer, the *x*, *y* and *z* values and results of the x^2, y^2 and z^2 calculations are retained. If a value hasn't changed from the previous sample then the stored squared value can be used. If none of the input values have changed, the entire calculation can be skipped since the result would be the same as the last time it was calculated. In the example, this approach can save considerable power even on most 32-bit micros since the square operation is performed in software.
- The approach above can save even more power when significant changes in input values are required before performing calculations. Going back to the accelerometer example again, if the alarm threshold isn't critical, the firmware could require a change in an input value of 0.05g or 0.1g before squaring the value, summing the squared values and comparing the result against the pre-calculated threshold value. This technique can be particularly useful when using sensors or A/D converters that have more precision than is required for you application. Of course, what constitutes a "significant change" will vary from application to application and applications that require a high degree of accuracy may not be able to use the approach at all.
- If your application requires trig functions (sine, cosine, etc.), using look-up tables may provide sufficient accuracy and save considerable power even on 32-bit micros.

In using some of these techniques you will be trading off accuracy for lower power. It is up to you to decide what is acceptable in this regard. Combining these techniques can make the decision whether

to perform the calculation a matter of a few dozen clock cycles compared to several thousands of clock cycles for the full calculation. If done smartly, only the portions of the full calculation that need to be done will be done based on the inputs changing from the previous samples.

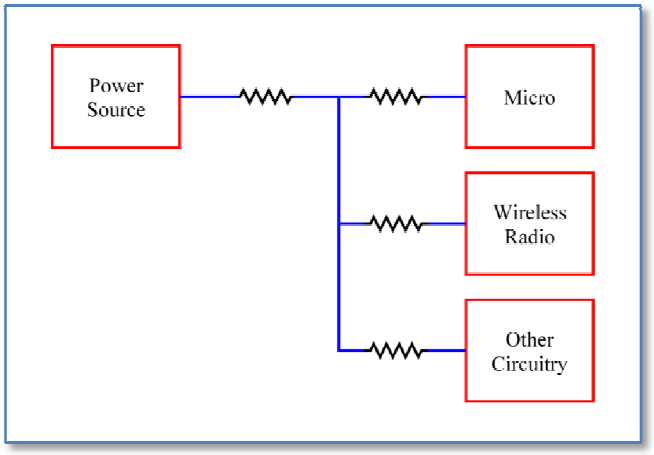
Library Functions

The generic library functions provided with a compiler are there for convenience and likely aren't very well optimized or have many options you don't need. The more options a library function supports the more likely it isn't power efficient. For example, a "printf" library function can waste many clock cycles checking for formatting options you aren't using (and take up more than 8KB of program space). Generic library functions can be useful for getting your code running quickly but you should consider replacing them with your own optimized functions.

Though not directly related to power savings, when you use the concepts and techniques presented here be sure to take the time to put comments in the code. Comments on WHAT the code is doing are helpful but comments on WHY the code is the way it is can be more important. When someone else cracks open your source code a year or two down the road to fix a bug or add a new feature, their first inclination will be to "fix" your strange looking code if they don't understand why it is like it is. The benefits to you will be two-fold. First, you won't have to drop what you're working on because your product's battery life really dropped for no apparent reason. Second, that person maintaining your code will tell your co-workers and manager "That Joe really knows his stuff!" instead of "Joe is a nice guy but he doesn't know squat about writing firmware."

Design for power measurements

- The most important thing in designing your board to allow for current measurements is to isolate the power supply rail to the micro and any other high power circuitry like a wireless radio module using separate current sense resistors for each major circuit as show below. You will also want to be able to measure the current at the point of power entry to the board. Unless you are counting nanoamps, this is usually good enough and isolating the power to each individual circuit just makes the board layout more difficult.



- If your board space allows for it, place a 2 pin “Berg” header in parallel with a current sense resistor. For normal operation you can jumper across the header and then remove the jumper for taking current measurements. After your development is completed it is easy to remove the jumper and current sense resistor in the board layout.
- If you can, it is a good idea to connect relevant digital signals to test pads or even a test header in order to provide scope triggers and context to current waveforms. It can be very difficult to trigger a scope on the current waveform for a specific event, particularly if your product has more than a few power states.
- You may also consider using spare GPIO pins to indicate certain events or conditions in the firmware. For instance, if you are using a wireless radio module and there are no digital signals that indicate when the radio is transmitting or receiving, using two GPIOs so the firmware can provide these indications can be very helpful when trying to capture the current waveforms for those events.

The Challenge Ahead

You've checked out your prototype and the firmware is working so now you want to take some current measurements and determine the power consumption of your product. Before getting your scope setup to start taking these measurements, you should be aware of several issues with this traditional method of taking current measurements, particularly with modern microprocessor based, software driven electronics:

- The power consumed by a typical battery powered product can vary from less than a microamp in a deep-sleep mode to tens or hundreds of milliamps when fully active. This wide dynamic range is nearly impossible to deal with using the traditional method. A sense resistor sized to give enough voltage drop to measure microamps will drop enough voltage to prevent the circuit from operating when the current changes to the milliamp range.
- Even at power levels of a few milliamps, the voltage drop across the sense resistor can be so low that it is difficult to measure and accurately assess the information presented on an oscilloscope screen. With voltage drops of a few millivolts the accuracy of the measurement can be impacted by the resolution and accuracy of most oscilloscopes.
- A multi-meter can provide more accurate steady-state measurements than a scope but it simply reports a pseudo-average current reading, only taking measurements several hundred times per second at best.

If trying to take current measurements isn't challenging enough, trying to determine power consumption over time can seem nearly impossible.

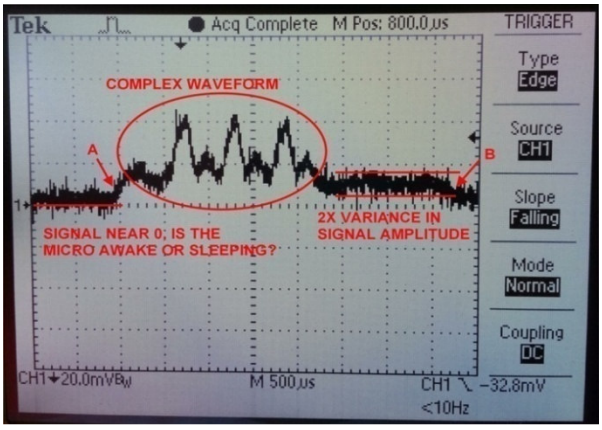
- Current and power consumption levels can be very transitory, changing rapidly and remaining steady for periods as short as a few microseconds. The power consumption can also vary by up to several orders of magnitude at these high rates of change. These factors make for a very complex waveform that you must analyze. While you may be able to visually analyze this complex waveform on an oscilloscope screen, the result won't provide an accurate measure of power consumed over time.
- A simple product like an MP3 player will have several power level states while something as sophisticated as a cell phone or laptop computer can have dozens of power level states. These states reflect what the user is doing with the device and can typically last from a few seconds to several hours. Each state may have dozens of different power consumption levels as noted in the previous bullet points.
- The traditional sense resistor and oscilloscope method is strictly a manual process. Even if you could accurately measure instantaneous current levels, it is nearly impossible to get a true measurement of power consumed over the time span of an event with anything but the simplest current waveform.

The scope screen shot with the current waveform on the following page illustrates some of these issues. This waveform was taken on the battery voltage of a Bluetooth Low Energy device with the sense resistors selected for 1mV equals 1mA. The event that produced this waveform occurs 10 times per second for as long as 30 seconds. This means that any error in the power consumption calculation for this event will be multiplied by as much as 300 for the series of events.

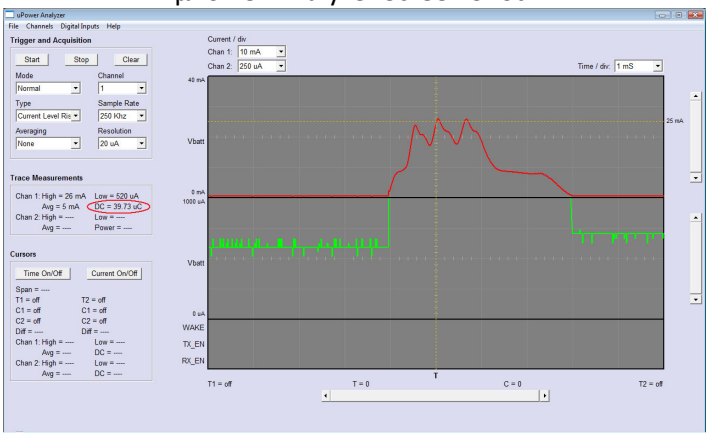
In addition to the issues noted on the screen shot, there are also several less obvious issues. The time between points “A” and “B” varies from about 3.5mS to 7mS, further complicating the process of making power consumption calculations. Additionally, this trace was captured using settings at the limits of the scope’s resolution and display capability. The result was a distorted and erroneous waveform. The three circled peaks appear to reach a little over 40mA but in reality they are only 26mA to 28mA. Because of this, the most accurate possible analysis of this waveform would still produce a power consumption measurement with considerable error.

CMicrotek is developing the µPower Analyzer™ and µPower Probe™ to address these issues. The µPower Probe is similar in function to traditional scope current probes but with much higher resolution, much lower minimum current ranges and the wide dynamic current range to deal with most typical battery powered products. The µPower Analyzer has similar performance specs plus the ability to take true power consumption measurements over time.

Scope Screen Shot



µPower Analyzer Screen Shot



Summary

- It is hard to emphasize it enough, the micro you select can set you up for power efficient operation or prevent you from obtaining low power consumption and long battery life. You have to read the micro datasheet and really learn what its capabilities are in its power saving modes. Keep in mind the most expensive mistake you can make in terms of power, development time and dollars is selecting the wrong micro for your product because you either start over to achieve acceptable power consumption or live with the results and hope your product doesn't fail to become a marketplace success because of it.
- In general, more modern micro designs can be significantly more power efficient than older designs. Don't just assume an 8-bit micro uses less power than a 32-bit micro. Many factors unique to your firmware implementation will ultimately determine how power efficient your firmware is.
- It seems counter-intuitive but for most types of products, to minimize power consumption you have to focus on what the firmware spends the most time doing, which usually isn't what the product's main function is. Most firmware based products spend most of their time waiting, waiting for user input or waiting for time to check some condition.
- When possible, let the hardware in your micro do the work instead of doing it in firmware. DMA controllers, UARTs with FIFOs and A/D converters that can autonomously do a series of conversions are a few examples of how hardware can do the work while the micro is sleeping.
- Every clock cycle your code executes uses power. Eliminating just one extra instruction or one extra memory access in a loop that executes hundreds of times a second can represent a significant percentage power savings. If your code is written in C, for ultimate power savings you may want to use assembly language for the most frequently executed portions of the program (and be careful with "in-line assembly language").
- It really pays to review the assembly code your compiler generates to learn what the compiler does efficiently and what to avoid if you can. Never assume your compiler will do things the same way you would if you were writing the assembly language.
- The more complicated your power distribution design the more traps you have to be careful about. Something as simple as a capacitor value or where a pull-up resistor is connected can cause a significant increase in power consumption (and maybe even lead to products failing in the field).
- Even a high-end oscilloscope may not be adequate for taking accurate current measurements with the wide ranging current levels found in today's power managed products. Specialized equipment is required for ultra-low current measurements and accurate measurements of power consumption over time.

