# Designing Low-Energy Embedded Systems from Silicon to Software

## Part 2 – Software Decisions

### Introduction

Low-energy system design requires attention to non-traditional factors ranging from the silicon process technology to the software that runs on microcontroller-based embedded platforms. Closer examination at the system level reveals three key parameters that determine the energy efficiency of a microcontroller (MCU): active power consumption, standby power consumption and the duty cycle, which determines the ratio of time spent in either state and is itself determined by the behavior of the software.

A low-energy standby state can make an MCU seem extremely energy efficient, but its true performance is evident only after taking into account all of the factors governing active power consumption. For this and other reasons, the tradeoffs of process technology, IC architecture and software construction are some of the many decisions with subtle and sometimes unexpected outcomes. The manner in which functional blocks on a microcontroller are combined has a dramatic impact on overall energy efficiency. Even seemingly small and subtle changes to hardware implementation can result in large swings in overall energy consumption over the lifetime of a system.

*Part 1 of this two-part article discussed chip-level design considerations that must be considered to achieve the lowest possible power consumption at the silicon device level.*

### *Part Two: Software Decisions*

### Performance Scaling

Implementing energy-efficient embedded applications relies on software design that uses hardware resources in the most appropriate way. What is appropriate depends not only on the application but also on the hardware implementation. Likewise, the more flexible the hardware in terms of CPU, clock, voltage and memory usage, the greater the potential energy savings the developer can achieve. Hardware-aware software tools provide the embedded systems engineer with greater awareness of what further savings are achievable.

One option is to employ dynamic voltage scaling, as shown in Figures 1 and 2. This technique is made possible by on-chip dc-dc converters and performance-monitoring circuits, which provide the ability to reduce the supply voltage when the application does not need to execute instructions at the highest speed. Under these conditions, the system operates with reduced power consumption. The benefits that can be achieved are a function of input voltage and can vary over the life of a product. The following figures show the relative differences between no voltage scaling (VDD fixed), static voltage scaling (SVS) and active voltage scaling (AVS).

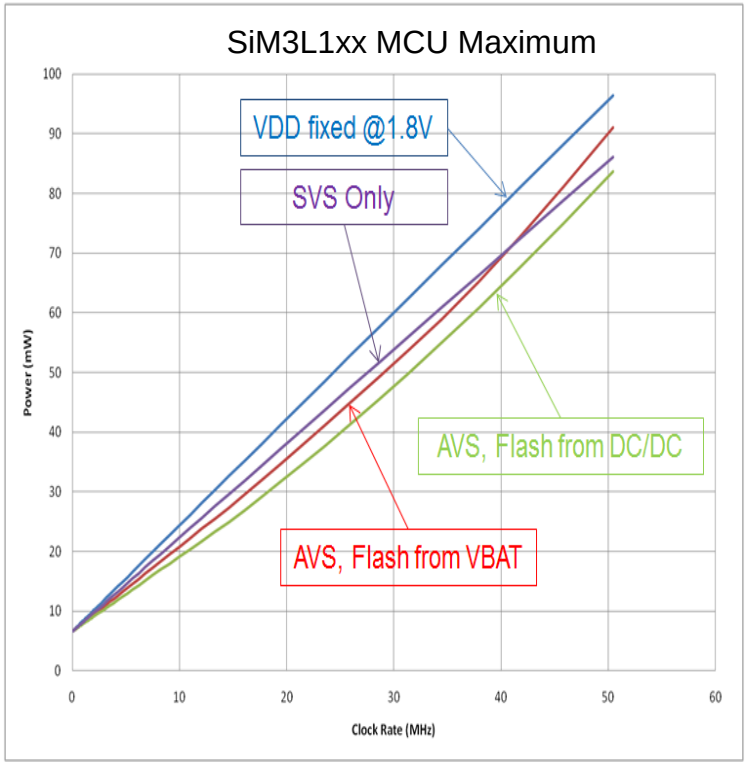


**Figure 1. Effects of Voltage Scaling with VBAT = 3.6 V**

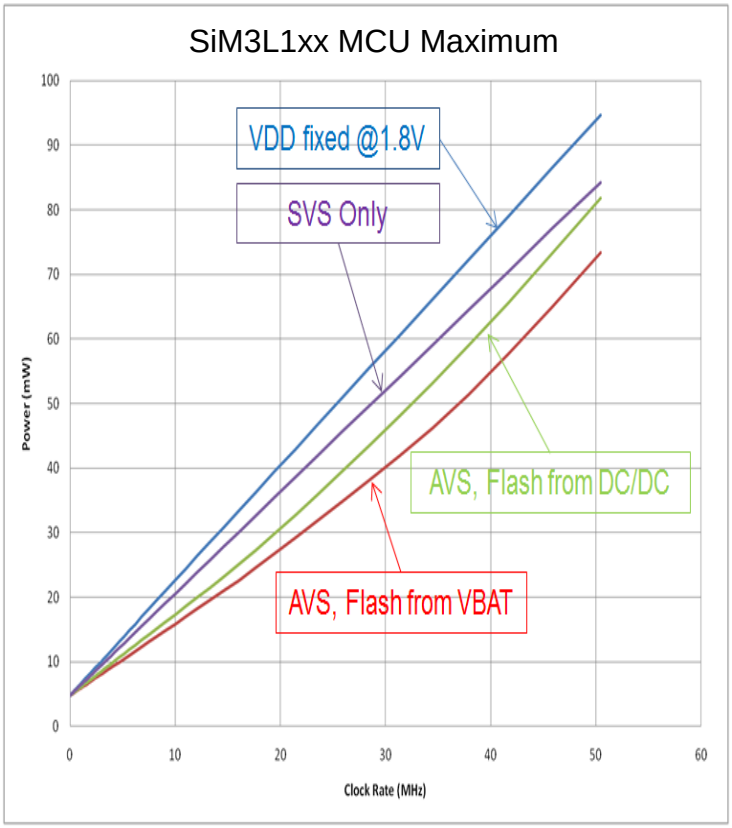


**Figure 2. Effects of Voltage Scaling with VBAT = 2.4 V**

An interesting artifact of AVS is that the AVS strategy can change depending on the input voltage to the system. In this example, when the input voltage is 3.6 V, it is more efficient to power the internal logic as well as the flash memory from a high-efficiency internal dc-dc converter. However, as the input voltage falls (i.e., battery discharge over product life cycle), it is more efficient to power the flash memory subsystem from the input voltage directly because the internal logic can operate at lower voltages than the memory. For example, the new SiM3L1xx low power 32-bit microcontroller family from Silicon Labs has a flexible power architecture with six separate and variable power domains that enables this kind of dynamic optimization.

Typically, CMOS logic circuits will operate more slowly as their voltage is reduced. If the application can tolerate lower performance (often the case when dealing with communications protocols that demand data be delivered no faster than a certain standardized frequency), then the quadratic reduction in energy consumption with lower voltage can provide large energy savings. Leakage provides a lower limit on voltage scaling. If each operation takes too long, leakage will begin to dominate the energy equation and increase overall energy consumption. For this reason, it can make sense to execute a function as quickly as possible and then put the processor into sleep mode to minimize the leakage component.

Take, for example, a wireless sensor application that needs to perform a significant amount of digital signal processing (DSP), such as a glass-breakage detector. In this example, the application uses a Fast Fourier Transform (FFT) to analyze the vibrations picked up by an audio sensor for the characteristic frequencies generated by glass shattering. The FFT is relatively complex, so executing it at a lower frequency governed by a reduced voltage is likely to increase leakage substantially, even in older process technologies. The best approach, in this case, is to execute at near maximum frequency and then return to sleep until the time comes to report any findings to a host node.

The wireless protocol code, however, imposes different requirements. Radio protocols have fixed timings for events. In these cases, the protocols can be handled entirely in hardware. It makes more sense to reduce the processor core's voltage. Therefore, the code needed for packet assembly and transmission runs at a speed appropriate to the wireless protocol.

The addition of hardware blocks such as intelligent direct memory access (DMA) can further change the energy trade-offs. Many DMA controllers, such as the one provided by the native ARM® Cortex™-M3 processor, require frequent intervention from the processor. However, more intelligent DMA controllers that support a combination of sequencing and chaining allow the processor to compute packet headers, encrypt data, assemble packets, and then hand over the work of passing the packets at appropriate intervals to the memory buffers used by the radio front-end. For much of the time that the radio link is active, the processor can sleep, saving a significant amount of energy.

## Memory Usage

With modern 32-bit microcontroller devices, software engineers have a high degree of freedom in the way memory blocks are used. Typically, the MCU will provide a mixture of non-volatile flash memory for long-term code and data storage along with static random access memory (SRAM) to hold temporary data. In most cases, the power consumption of flash memory accesses will be higher than those made to SRAM. In the normal usage case, flash memory reads exceed SRAM reads by a factor of three. Flash memory writes, which require entire blocks to be erased and then rewritten using a lengthy sequence of relatively high-voltage pulses, consume even more power; however, for most applications, flash write operations are infrequent and do not materially affect the average power consumption.

A further factor in flash-memory power consumption is how accesses from the processor are distributed. Within each block of flash memory there are multiple pages, each of which can be up to 4 kB in size. To support any accesses, each page has to be powered-up; any unused pages can be maintained in a low-power state.

If a regularly accessed section of code straddles two flash pages rather than being contained within one, the energy associated with instruction reads will increase. Reallocating memory to place frequently

accessed sections of code and data within discrete pages can result in sizeable savings in power consumption over the lifetime of a battery charge with no changes to the physical hardware.

It often makes sense to copy functions that are used more frequently into on-chip SRAM and read their instructions from there rather than from flash, even though this appears to use the memory capacity less efficiently. The benefit in battery life can easily make up for the slightly higher memory consumption.

## Code Optimization

Energy optimization can also upend traditional ideas of code efficiency. For decades, embedded systems engineers have focused on optimizing code for memory size except when performance is critical. Energy optimization provides an altogether new set of metrics. An important consideration is usage of the on-chip cache that is generally available to 32-bit microcontroller platforms.

Optimization for code size enables retention of more of the executable in cache, which improves both speed and energy consumption. However, function calls and branches that are used to reduce the size of the application through the reuse of common code can result in unintended conflicts between sections of the code for the same line in the cache. This can result in wasteful ‘cache-thrashing’ as well as multiple flash page activations when the instructions need to be fetched from main memory.

For code runs frequently during the lifetime of the product, it makes sense for it to be sufficiently compact to fit into the cache but not to branch or call functions. Consider a smoke alarm; even if the alarm triggers once a week (perhaps from excess smoke caused by activity in the kitchen), that is only 520 events out of 315 million during the alarm’s 10-year life. The vast majority of the time, the code only takes a sensor reading, finds that the threshold has not been exceeded and then puts the processor core back to sleep until it is woken by the system timer.

Out of all the sensor readings that the alarm takes, less than 0.0002 percent will result in the execution of alarm-generating code. The remaining 99.9998 percent of code execution will be of the core sensor-reading loop. Ensuring that this code is run in a straight line out of cache can be the key to minimizing energy usage. Because it runs so infrequently, the remaining code can be optimized using more traditional techniques.

## Tools for Energy Efficiency

Tool support is vital for maximizing the energy efficiency of an MCU platform. The ability to allocate functions to discrete pages of flash memory requires a linker that understands the detailed memory map of each target microcontroller. The linker can take developer input on whether blocks can be allowed to cross page boundaries and generate a binary that is optimized for the most energy-efficient use of non-volatile storage. In principle, this code is also used to ensure that functions and data are placed in such a way that the most commonly executed ones do not clash over cache lines. This level of detail can be achieved much more easily when the tools are provided by the MCU vendor (who knows the memory layout and power requirements of each target platform). This is far more difficult for a third-party vendor to achieve.

The MCU also has a detailed understanding of how the different peripherals and on-chip buses are organized. This knowledge can be applied in tools to guide the engineer in making choices that do not waste power. The graphical AppBuilder environment developed by Silicon Labs is one such example, as shown in Figure 3. This tool makes it possible to define the framework for an application by dragging and dropping peripherals onto a canvas.

AppBuilder can look at the peripheral setup and determine whether energy-saving changes are possible. For example, if a user has pulled a UART into the application and set its speed to 9600 baud, the tool will view the peripheral bus of the UART and determine the appropriate setting. The ARM Peripheral Bus (APB) used to host blocks, such as UARTs and analog-to-digital converters, can run at up to 50 MHz. In

this instance, this speed is far higher (and will consume more energy) than is necessary; so, the tool asks if the user wants to reduce the APB's data rate to a level that is more appropriate.

In addition, AppBuilder software provides the engineer with other application-specific information on power consumption. Using a simulation of the target MCU (again made possible by a detailed understanding of the silicon features), the tool can provide an interactive histogram of estimated current not just for the entire application but for the processor and each peripheral.

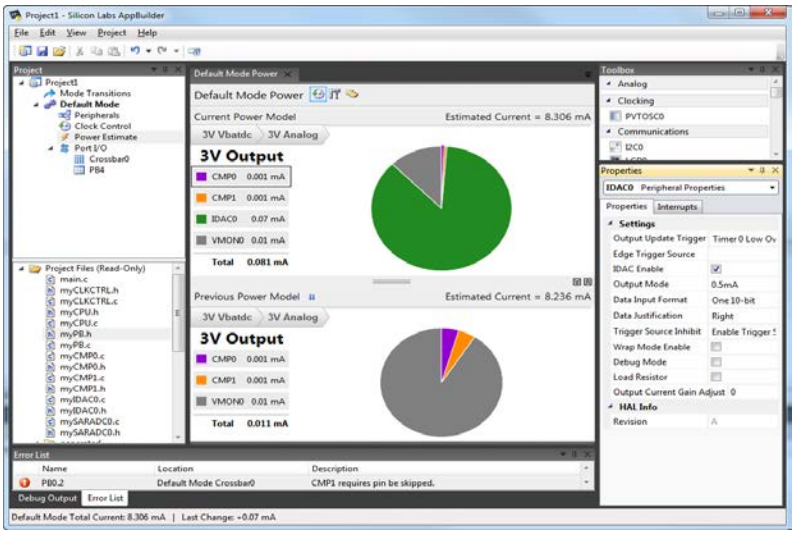


**Figure 3. Screen Image of AppBuilder with Power Profiler Tool**

Development tools will evolve to become more "power-aware." Traditionally, debug features such as breakpoints have been set on events (i.e., memory reads and writes). In the future, it is conceivable that breakpoint support will evolve to handle power-related issues. For example, if power consumption at a particular point or the integrated energy since the last sleep state exceeds a target, the debugger will trigger and show which parts of the application are consuming higher-than-expected amounts of power (e.g., code that straddles a flash-page boundary may be running more frequently than expected). Higher-than-expected consumption and information on the code's position in the memory map provide vital clues to help the software engineer take appropriate action.

## Conclusion

Low-energy system design is a holistic process that is enabled by choosing a combination of the right silicon, software and development tools. By mastering the relationship between each of these variables, systems engineers can develop higher performance and more energy-efficient embedded systems that stretch the limits of battery-powered applications.

# # #