从硅芯片到软件 - 设计低能耗嵌入式系统

第二部分 软件设计原则

简介

设计低能耗系统时，我们需要关注一些非传统因素，这些因素涉及范围从硅芯片生产工艺技术，到基于单片机的嵌入式平台上所运行的软件。通过对系统层面的深入分析，本文讨论决定 MCU 能效的三个关键参数：工作模式功耗、待机模式功耗和占空比（他决定各种状态下所占用的时间比率，由软件自身来决定）。

低能耗待机状态使 MCU 看上去具有超高能效，但事实是，只有考虑了控制工作模式功耗的所有因素后才能决定 MCU 的能效状况。总之，对于处理工艺、IC 架构和软件结构选择的权衡是十分微妙的决定因素，有时会出现意想不到的结果。此外，单片机上功能模块相结合的方式对整体能耗加以动态影响。即使对硬件实现看似小而轻微的改变，都可能会导致系统运行周期中整体能耗的大幅波动。

本文的第一部分讨论了在硅芯片级别上，实现最低功耗的芯片级设计原则。

第二部分：软件设计原则

性能扩展

如何实现高能效嵌入式应用，这依赖于软件设计，而软件设计又要以最合适的方式使用硬件资源，其中应用和硬件实现同样重要。同样的，处理器、时钟、电压和内存利用等硬件的灵活性越大，开发人员可实现的节能潜力越大。此外，能感知硬件的软件工具可以帮助嵌入式系统工程师辨别出能实现何种节能。

有一种方法是采用动态电压缩放，如图 1 和图 2 所示。此技术通过片上 DC-DC 变换器和性能监视电路来实现，当系统不需要以最高速率执行指令时，可以降低供电电压。由此，系统运行处于低功耗。好处在于其成为输入电压的函数，并且能在产品生命周期中改变。下图显示固定电压（VDD 固定）、静态电压缩放（SVS）和主动电压缩放（AVS）之间的相对差异。

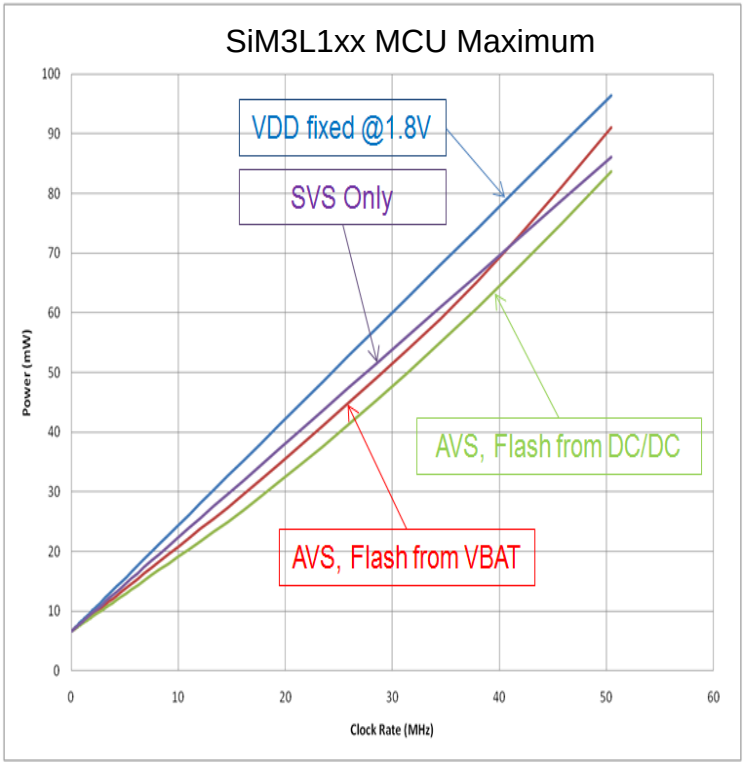

图 1：VBAT=3.6V 时电压缩放效率

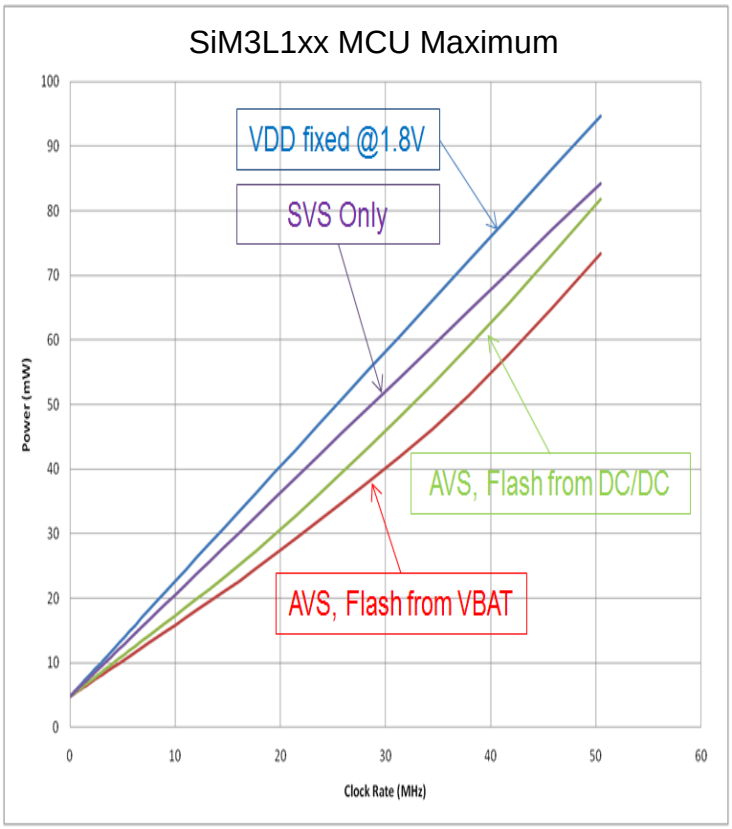

图 2：VBAT=2.4V 时电压缩放效率

AVS 有一个令人感兴趣的现象，AVS 策略依赖于输入到系统中的电压而改变。在这个示例中，当输入电压为 3.6V 时，采用内部 DC-DC 变换器为内部逻辑（也包括 Flash 存储器）供电更为高效。然而，当输入电压降低以后（例如电池在产品生命周期中放电），采用输入电压直接为 Flash 存储器子系统供电是更高效的方法，因为此时与存储器相比，内部逻辑能够在更低的电压下运行。例如来自 Silicon Labs 的新型 SiM3L1xx 低功耗 32 位单片机系列产品具有灵活的供电架构，支持六个独立和可变的供电域，可支持动态优化策略。

通常情况下，当电压降低时 CMOS 逻辑电路操作变慢。如果应用可以容忍更低性能（一般当处理通信协议时，所需的数据速率不超过特定标准化频率），那么低电压所带来的能耗降低则更加省电（以平方倍数降低）。漏电损耗提供电压缩放的下限。如果操作花费太多时间，那么漏电损耗将开始主导能耗方程式，并且增加整体功耗。出于这个原因，为了使漏电组件功耗最小化，应该尽快执行操作，然后使处理器进入休眠模式。

假设一个无线传感器应用，其需要进行大量的数字信号处理（DSP），例如玻璃破碎检测器。在这个示例中，使用快速傅里叶变换（FFT）分析由音频传感器提取的变动（特性频率由玻璃破碎产生）。FFT 是一个相对复杂的算法，因此通过降低电压而得到的低频下运行该算法，可能显著增加漏电损耗，即使采用旧的生产工艺技术也一样。在这种情况下，最好的方法是在接近最大频率下运行，然后迅速返回到休眠状态，一直到需要向主机报告时再运行。

然而，无线协议代码要满足不同的需求。无线协议有固定的事件时序。在这些情况下，协议可以完全由硬件处理。从而降低处理器内核电压。因此，包组装和传输代码可以运行在适合于无线协议的速率。

附加的硬件模块，例如智能直接内存访问（DMA），能够进一步改变能耗。许多 DMA 控制器，例如由 ARM® Cortex™-M3 处理器提供的 DMA 控制器，与处理器交互频繁。然而，更多的智能 DMA 控制器同时支持顺序和链接，允许处理器计算包头、加密数据、构建包，然后以适当的间隔时间把数据包交付到由射频前端使用的内存缓冲区中。因为射频链路的大多数时间是处于活动状态，处理器能够进入休眠模式，从而节省大量的能量。

存储器的利用

采用现代 32 位单片机器件，软件工程师能比较自由的使用内存区块。通常，单片机提供混合存储方案，非易失性 Flash 存储器用于长期代码存储，而静态随机访问存储器（SRAM）用于保留临时数据。大多数情况下，访问 Flash 存储区所产生的功耗大于访问 SRAM 存储区所产生的功耗。在正常使用过程中，Flash 存储区读取时间是 SRAM 读取时间的两倍。Flash 写操作需要先把整块区域删除，然后使用较长的相对高压脉冲序列重写，消耗很多能量；然而对于大多数应用来说，Flash 写操作很少发生，不会显著影响平均功耗。

影响 Flash 存储器功耗的另一个因素是如何分配处理器访问方式。Flash 存储器的每个区块由多个页面组成，通常最大为 4kB。为了支持任意访问，所有页面都要上电；任何未使用的页面都维持在低功耗状态。

如果经常访问的代码段横跨两个 Flash 页面而不是在一个页面中，那么指令读取相关的能耗将增加。为频繁访问的代码和数据重新分配非关联的页面能够节省大量电池能耗，且不需要对物理硬件进行改变。

通常明智的做法是把频繁使用的功能代码复制到片上 SRAM，从 SRAM 而不是从 Flash 中读取这些代码指令，虽然这样看起来存储器效率有所降低。电池寿命的延长弥补了略高的内存消耗。

代码优化

能耗优化也颠覆了传统观念上的代码效率。几十年来，嵌入式系统工程师都集中在为内存大小而优化代码。能耗优化提供了一套全新指标，一个重要的考虑因素是片内缓冲区的利用，他通常存在于 32 位单片机平台。

代码大小的优化使缓冲区中可以保留更多可执行代码，从而改善速度和能耗。然而，重复使用的公共代码，包括过去常被用来减少应用程序大小的函数调用和分支，会在高速缓冲区中相同行上导致意想不到的冲突。当指令从主存储区中提取时多个 Flash 页面激活时，将导致高速缓冲区冲突以及多 Flash 页面激活。

对于在产品整个生命周期中经常运行的代码，好的办法是把除了分支或调用函数以外的代码压缩到缓冲区。比如烟雾报警器。即使每周触发一次报警（可能由厨房中的过量烟雾引起），在报警器 10 年寿命 3.15 亿次检查中仅有 520 次事件发生。大部分时间中，代码仅仅执行传感器读取，检查是否超出门限，然后处理器内核处于休眠状态直到被系统定时器再次唤醒。

在报警系统执行的所有传感器读取之外，不到 0.0002%的几率会导致报警生成代码执行。剩下 99.9998%的时间处于核心传感器读取循环。使能耗最小化的关键在于，确保报警生成代码不在缓冲区执行序列中运行。因为他较少运行，因此可采用更传统的技术优化这类代码。

能效工具

为了使 MCU 平台的能效最大化，工具支持至关重要。把功能分配到 Flash 上非关联页面中，这需要连接器清楚每个目标单片机的详细内存映射。连接器能够根据开发者的输入来判别模块是否可以跨越页面边界，并为非易失性存储生成最高能效的二进制代码。原则上，这个代码也用于确保函数和数据被这样放置 - 最常用的执行函数不会在缓冲区中产生

冲突。由于 MCU 供应商所提供的工具清楚知道每个目标平台的存储区布局和功耗需求，以上细节层次更容易实现。而对于第三方供应商来说则相当困难。

MCU 非常清楚如何组织不同的外设和总线也非常详细的理解。这类知识能够应用在工具中指导工程师进行选择，且不会浪费能量。由 Silicon Labs 开发的图形化 AppBuilder 环境就是一个典型示例，如图 3 所示。此工具可以通过拖拽外设来为应用定义软件框架。

AppBuilder 能够查看外设设置并决定是否可以进行节能调整。例如，如果用户把串口添加到应用，并设定波特率为 9600，则工具将检查 UART 外设总线并进行适当的设定。ARM 外设总线（APB）用于主机模块（例如 UART 和模数转换器）能够运行在最大 50MHz 速率下。在这个示例中，该速率远远高于（并会消耗更多的能量）所需的速率；因此工具要求用户是否想减少 APB 的数据速率到一个更加适合的级别。

此外，AppBuilder 软件为工程师提供与功效相关的特殊应用信息。使用目标 MCU 的仿真（也能够通过详细的硅芯片特性理解来实现），工具可提供交互式图表以显示电流消耗，不仅仅对于整个应用，同时也用于处理器和每个外设。

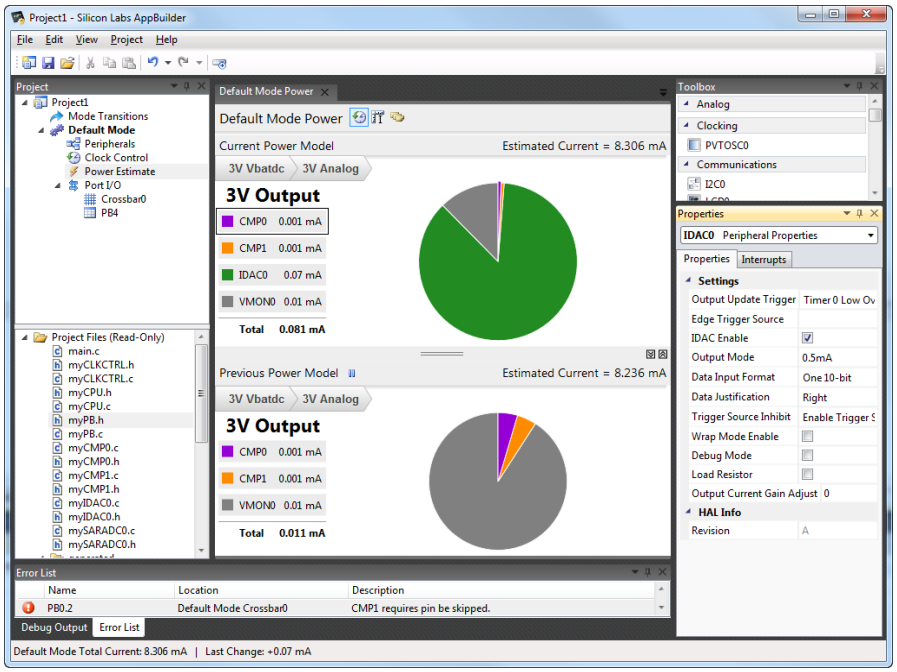

图 3：具有功耗感知工具的 AppBuilder 界面截图

开发工具将会演变并具有更多“功耗感知”能力。传统上，调试特性（例如断点）被设置为事件（例如内存读写事件）。未来，断点支持会用作处理功耗相关的问题。例如，如果在特殊门限点或自上次休眠状态后所产生的累积功耗超过目标值，调试器将触发，并显示出在应用中哪些部分功耗超过预期（例如，跨越 Flash 页面边界的代码可能更频繁的超过预计）。比预计更高的功耗以及在内存映射中代码位置上的信息，可以提供关键线索，从而帮忙软件开发人员采用适当措施。

结论

低能耗系统设计是一个整体设计过程，需要选择合适的硅芯片、软件和开发工具。通过掌握这些变量因素之间的关系，系统工程师可以开发出更高性能和能效的嵌入式系统，突破电池供电型应用的诸多限制。

# # #