Distribution
According to Need

不用的模块可以关闭么？
CPU 和设备可以工作在不同的时钟下么？
可以不打断系统工作而改变时钟频率么？

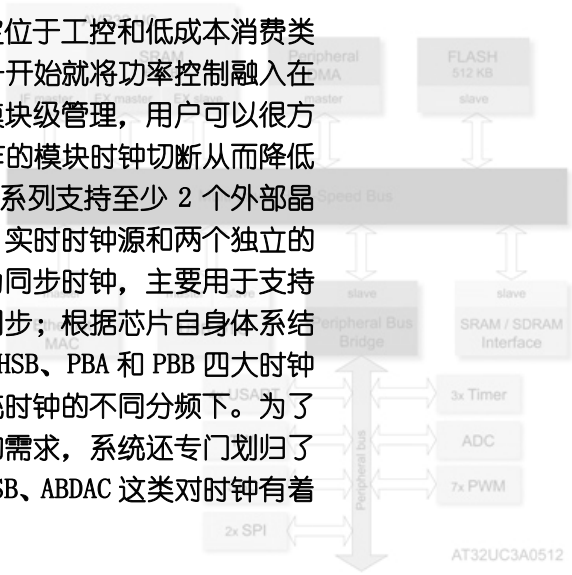
——**All that is OK**！

>> Power Manager

第三章

本章引言

AVR32 UC3 系列作为一款定位于工控和低成本消费类电子产品的 32 位微处理器，从一开始就将功率控制融入在芯片的设计中。借助对时钟的模块级管理，用户可以很方便的将包括 CPU 在内暂时不工作的模块时钟切断从而降低功耗。在时钟的配置方面，UC3 系列支持至少 2 个外部晶振/时钟源；一个独立的 32Khz 实时时钟源和两个独立的 PLL 锁相环。系统主时钟又称为同步时钟，主要用于支持 CPU 工作以及内部各模块间的同步；根据芯片自身体系结构的特点，系统被划分为 CPU、HSB、PBA 和 PBB 四大时钟域，它们可以工作在对同一系统时钟的不同分频下。为了满足某些模块对特定时钟频率的需求，系统还专门划归了一类普通时钟，用于支持类似 USB、ABDAC 这类对时钟有着特殊频率要求的模块。



AVR32.UC3.PM.Summary 综述

7.1 PM 模块简介

AVR32 UC3 系列有一个强大而功能又多样化的电源管理模块 **PM**，主要分为时钟频率，复位 **BOD** 以及功耗控制模式控制三大功能，**PM** 支持以下特性：

A、内部集成超低功耗 **115.2KHz RC** 振荡器，

B、由 **RC** 振荡器分频提供 **32.768KHz** 实时时钟，并支持外部 **32.768KHz** 晶振

C、支持 **2** 组 **450kHz** 至 **16MHz** 的外部晶振

D、支持 **2** 组频率 **80-240MHz** 的 **PLL** 锁相环模块

E、可提供宽频率的时钟信号输出用于信号同步

F、独立的 **USB** 以及 **ABDAC** 模块频率提供模式

G、支持多种软硬件复位模式

H、集成可配置 **BOD** 模块

I、可迅速改变 **CPU HSB** 总线，**PBA** 总线和 **PBB** 总线的时钟频率，并调整对应外设的工作频率

J、可独立完全关闭不使用的模块

K、支持 **6** 个睡眠模式

AVR32.UC3.PM.Feature 产品特性

7.2.1 灵活的时钟频率提供方式

AVR32 UC3 系列提供灵活的时钟频率工作模式，根据实际应用的需求，**AVR32 UC3** 集成一组超低功耗 **115.2KHz RC** 振荡器，支持两组 **450kHz** 至 **16MHz** 的外部晶振以及 **2** 组频率 **80-240MHz** 的 **PLL** 锁相环模块。并且支持 **32.768KHz** 实时时钟，在要求更高的环境下可以外接高精度 **32.768KHz** 实时晶振。

通过这些不同的时钟来源，**PM** 提供同步时钟（**Synchronous Clock**）以及普通时钟（**Generic Clock**）两类时钟信号。同步时钟（**Synchronous Clock**）主要提供 **CPU** 以及各级总线的时钟频率。芯片所有的时序逻辑都来自于同步时钟。普通时钟（**Generic Clock**）用来提供宽范围的时钟频率输出。用以需要特殊频率通讯设备的信号同步。同步时钟（**Synchronous Clock**）独立并且唯一，而普通时钟（**Generic Clock**）则支持多个通道不同频率的输出。另外，**USB** 以及 **ABDAC** 由内部的普通时钟通道提供独立的频率。

AVR32 UC3 系列通过两组外部晶振以及 **2** 组 **PLL** 锁相环模块的组合为特定的频率应用提供的可能性。比如一个 **USB MP3** 播放器应用。同时需要三种频率，**CPU** 需要 **66MHz**，**USB** 需要 **48MHz**，音频运放设备则需要 **12.288MHz** 的频率。而一般的芯片，因为只提供一个时钟源，如果需要用到 **USB CPU** 主频只能固定在 **8MHz** 的倍数频率上，往往不能全速运行，而音频设备的频率更加无从谈起。而 **AVR32 UC3** 系列因为多时钟源的结构，可以选择一个 **12M**

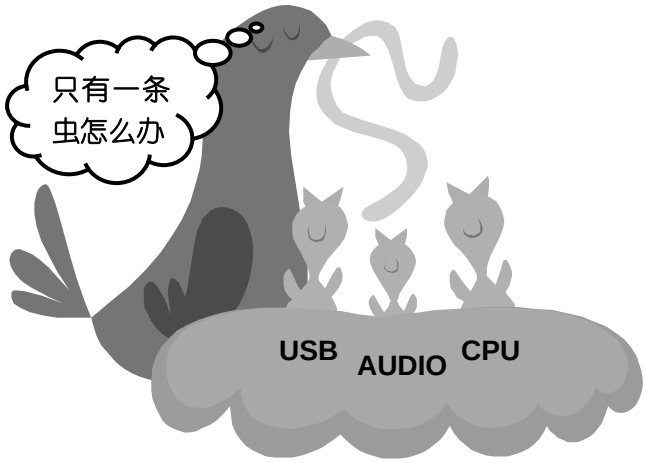


图 7.2.1 多时钟频率的必要性

晶振作为 **OSC0** 通过 **PLL0** 倍频到最高 **66MHz** 作为同步时钟（**Synchronous Clock**），并使用 **PLL1** 倍频至 **48MHz** 作为普通时钟（**Generic Clock**）提供给 **USB**，另外再设置一个 **12.288MHz** 晶振作为 **OSC1** 提供给音频设备，这样就很好的解决了时钟频率冲突的问题。

并且，**AVR32 UC3** 系列所有的时钟频率设置都是软件设置，可以在应用中再次更改运行频率以应对特殊的情况。

7.2.2 牢固的芯片运行保障措施

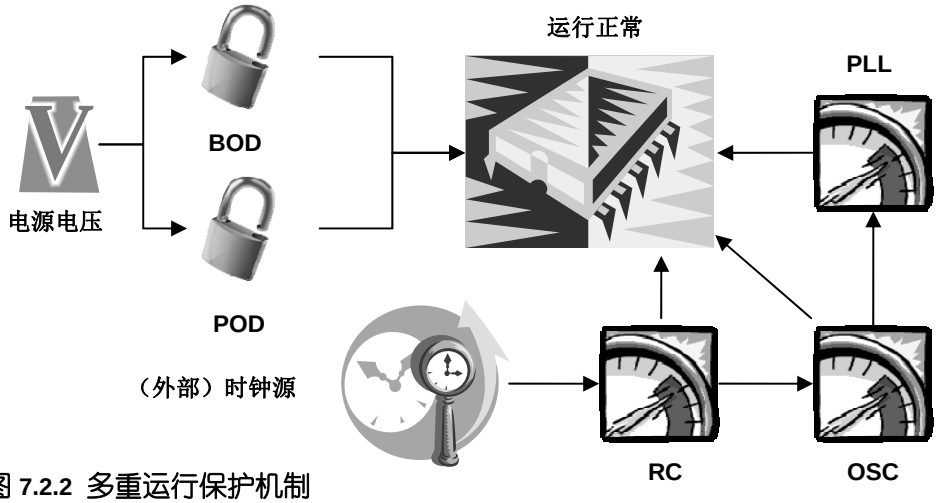


图 7.2.2 多重运行保护机制

AVR32 UC3 系列使用了多种策略保证能够安全的运行应用程序，芯片模块集成了掉电检测（**Brown-Out Detector**）以及上电检测（**Power-On Detector**）两个模块，实时监控 **AVR32 UC3** 系列的电源电压状况已保证芯片不会因为电压不稳定而造成的误操作。

AVR32 UC3 系列运行频率最高可至 **66MHz**，为了成功的运行至这个频率，**AVR32 UC3** 系列采用逐级软件设定工作频率的方式。系统复位后，芯片首先使用可靠的内部 **RC** 作为主时钟，然后再设定外部晶振，设定成功后再启动 **PLL** 模块，并且芯片带有侦错机制，如有相关频率发生模块设定不成功，芯片将不会进入应用的运行状态。如图 **7.2.2** 所示

7.2.3 精密的功耗控制

AVR32 UC3 系列以功耗控制著称，强调在最小的功耗下完成所需要的应用，并不只片面强调超低功耗，而制约应用。

首先，**AVR32 UC3** 系列支持 **6** 级低功耗模式，分别对应关闭不同的芯片模块以便实现超低功耗的需要。应用可以选择不同的唤醒方式快速调整芯片的运行状态。

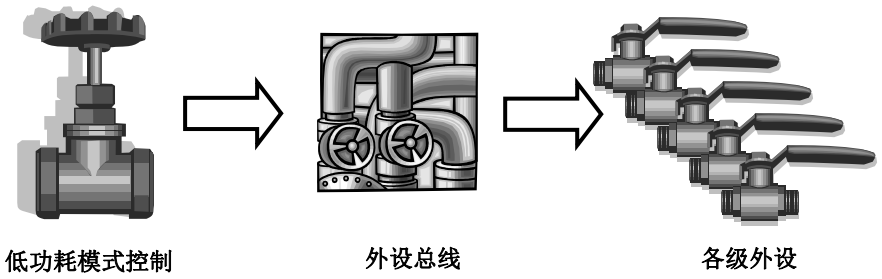


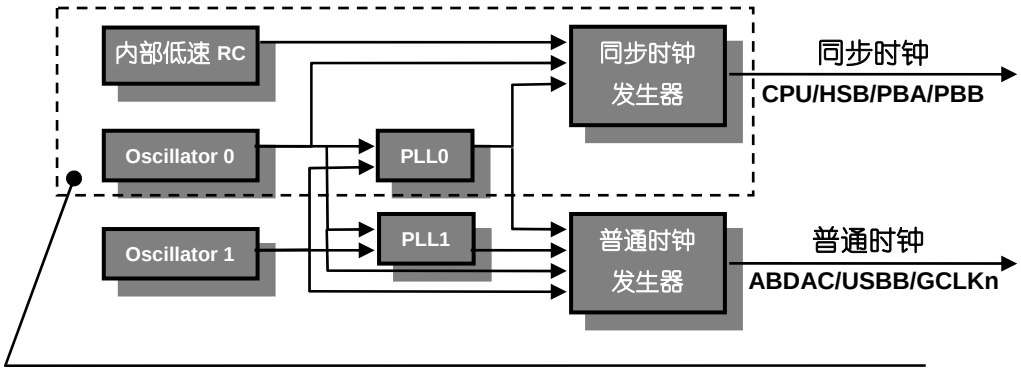
图 7.2.3 精细功耗控制

另外，为了实现紧密地功耗控制，**AVR32 UC3** 系列可以通过各级总线独立的调整各个芯片模块外设的运行频率，甚至可以实现对各个外设模块的独立掉电。从而实现在最低功耗的

要求下满足应用的需求，实现功耗应用最合理。如图 **7.2.3** 所示，假如 **6** 级低功耗模式是一个电源功耗控制的总阀门的话，各级总线就是各个支路上的小阀门，分别管理者相对应的模块。而各自独立的模块功耗控制模块就是一个系统中最小的调节部分，按需使用能源，没有任何的浪费，并且，所有的模块频率运行调整都可以在应用的运行中快速完成而不需要任何破坏程序运行的操作。

AVR32.UC3.PM.Interface 接口与功能概述

图 **7.3.1** 时钟系统模块关系结构图



注意：只有虚线框中包含的模块才可以作为同步时钟的时钟源。除了内部低速 115K RC 以外，Oscillator 0/1 以及 PLL0/1 都可以作为普通时钟的时钟源。

7.3.1 Oscillator 0、Osillator 1 和 32KHz Oscillator

AVR32 UC3 系列支持两个独立的外部时钟源通道 **Oscillator 0** 和 **Oscillator 1**，一般简称为 **OSC0** 和 **OSC1**。它们既可以通过匹配电容连接石英晶体振荡器（其连接方法如图 **7.3.2** 所示），也可以连接外部时钟 **Extern Clock**。时钟源的频率范围必须满足 **450KHz~16MHz**。

除了 **OSC0** 和 **OSC1** 以外，系统还支持一路独立的低功耗 **32KHz** 时钟源，称为 **32KHz Oscillator**。它们在晶振/外部时钟的连接方法上类似：

- 当 **OSC0**、**OSC1** 或者 **OSC32** 连接外部时钟时，**XIN**（**XIN32**）作为时钟信号的输入引脚，而 **XOUT**（**XOUT32**）则可以作为普通的 **GPIO** 来使用。这个过程中，**XIN**（**XIN32**）引脚功能的选择是自动而强制的，无须再设置 **GPIO**。

- 当 **OSC0**、**OSC1** 或者 **OSC32** 连接石英晶体振荡器时，引脚 **XIN** 和 **XOUT** 都将作为信号引脚，并需要串联一个匹配电容接地。这个过程中，引脚功能的选择是自动而强制的，无须再设置 **GPIO**。我们需要根据具体的晶振频率使用正确数值范围的电容。这里给定一种典型的电容计算方法：偏置电容的值 **C** 可以通过晶振的负载电容值 **CL** 和 **MCU** 内部的电容值 **Ci** 按照下面的公式计算获得：

C = 2 (CL - Ci)

其中，**CL** 的值可以从晶振的数据手册上获得，**Ci** 的值可以从芯片的数据手册上得到。需要注意的是，退耦电容应该尽可能的靠近时钟的信号引脚以避免不必要的干扰。

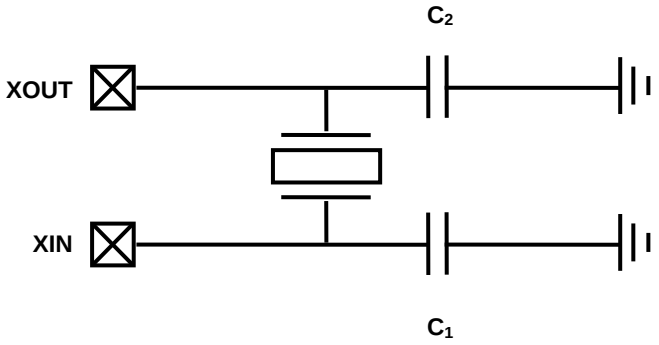


图 7.3.2 外部石英振荡器连接示意图

对 **OSCn** 来说，在确认外部连接以后，首先要通过寄存器设置选择连接的时钟源类型：是外部时钟还是石英振荡器。其中石英振荡器模式下还应根据具体的频率范围选择对应的工作模式。这些都通过设置寄存器 **OSCCTRLn** 的 **MODE** 位来完成的。

对于连接了晶振的情况，我们还应该根据晶振和匹配电容的参数选择设置晶振的启动时间（**Startup time**）。对启动时间的设置可以通过寄存器 **OSCCTRLn** 的 **STARTUP** 位来具体制定。

完成以上设置以后，就可以使能 **OSCn** 了。这一操作由设置 **MCCTRL** 寄存器的 **OSCnEN** 标志位来完成。同样，给该标志位清零将关闭对应编号的 **Oscillators**。**OSC32** 与 **OSCn** 一样，每次使能以后，都需要一个相对较长的时间来启动，而在时钟变得有效之前，寄存器 **POSCSR** 的 **OSCnRDY** 将被自动清零，直到 **OSCn**（**OSC32**）完成启动才会被再次置位。检测 **OSCn**（**OSC32**）再使能后是否有效是非常重要的。同时，由于晶振的启动非常耗时，因此如果允许应该避免将暂时无用的 **OSCn**（**OSC32**）关闭。

与**OSCn**相关的寄存器设置，可以查阅器件手册**Power Manager**章节的**UserInterface**小结。涉及到的寄存器有：**MCCTRL**、**OSCCTRL0**、**OSCCTRL1**。与**OSC32**相关的设置都在寄存器**OSCCTRL32**中。其它相关的寄存器还有**POSCSR**、**IER**、**IDR**、**IMR**和**IFR**。

7.3.2 锁相环 PLL

为了配合同步时钟（**Synchronous Clock**）和普通时钟（**Generic Clock**）对多样化时钟源的需求，**AVR32 UC3** 配备了至少两个锁相环，一般通称为 **PLLn**，其中 **n** 是锁相环的编号。

PLLn 的内部控制逻辑如图 **7.3.3** 所示。每个 **PLL** 都可以独立的选择 **OSC0** 或 **OSC1** 作为输入时钟源。对于每路输入 **PLL** 的时钟信号，都可以设置倍频因子（做乘法）和分频因子（做除法），并根据下面的公式计算出初步的 **PLL** 输出频率 **fvco**：

当 **PLLDIV** 大于 **0** 时（也就是除法因子不为 **0** 时）

fvco = (PLLMUL+1)/(PLLDIV) • fosc

当 **PLLDIV** 为 **0** 时

fvco = 2*(PLLMUL+1) • fosc

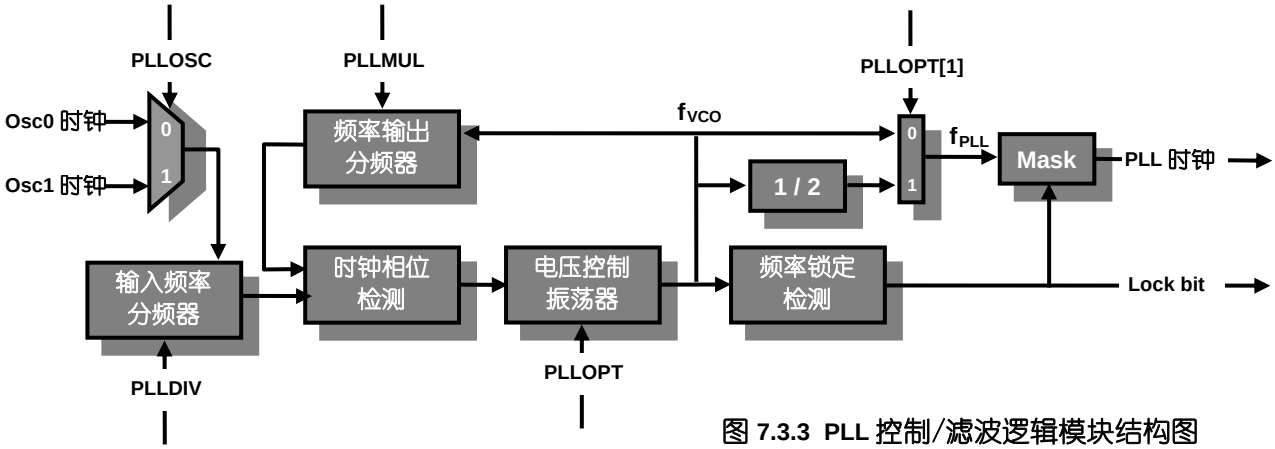


图 7.3.3 PLL 控制/滤波逻辑模块结构图

其中，**fosc** 表示 **PLLn** 的输入时钟频率，**PLLDIV** 表示分频因子，**PLLMUL** 表示乘法因子，**fvco** 表示 **PLL** 内部电压控制振荡器的输出频率。**fvco** 可以简单的理解为 **fosc** 在 **PLLn** 频率锁定以后经过倍频和分频操作的初步时钟输出。初步观察公式注意到：无论 **PLLMUL** 还是 **PLLDIV** 都可以取“**0**”值，分别表示不倍频和在 **PLLMUL** 基础上两倍频。

对 **PLLMUL** 和 **PLLDIV** 的取值范围，实际上有一定的限制：

首先，**PLLn** 的初步输出频率 **fvco** 必须满足 **80M~180MHz** 或者 **160M~240MHz** 两个范围的任意之一，考虑到 **OSC0** 和 **OSC1** 的输入时钟频率范围（**450KHz~16MHz**），当 **PLLDIV** 不为 **0** 时，**PLLMUL** 取值肯定不能低于 **4**（**80M / 16M = 5**）；当 **PLLDIV** 为 **0** 值时，**PLLMUL** 取值肯定不能低于 **2**（**(80M / 2)/ 16 = 2.5**）。

其次，由于 **PLLMUL** 和 **PLLDIV** 都只有 **4** 位二进制长度，能表示 **0~15** 以内的整数。当 **OSCn** 取 **16M** 时，**PLLMUL** 取最大值 **15**，则 **PLLDIV** 将可以取到取值范围的上限 **3**。也就是说，平时 **PLLDIV** 的取值范围只有 **0~3** 而已。而 **PLLMUL** 的取值范围是 **4~15**。

获得 **fvco** 以后，可以将 **fvco** 直接作为 **PLLn** 的时钟输出，也可以可以通过一个控制选项将 **fvco** 二分频以后输出。因此，我们实际可以得到的 **PLL** 频率输出范围在 **40M~240MHz** 之间。所有这些设置和选项都可以通过 **Software Framework** 提供的接口函数实现，而无须用户亲自对底层硬件操作进行干预。

设置 **PLLn** 的一般过程为：首先从 **OSCn** 中选择输入时钟源，设置 **PLLMUL** 和 **PLLDIV** 参数；其次，设置 **PLLn** 的工作模式，其中包括 **fvco** 的输出频率范围选择、是否对 **fvco** 二分频后再做频率输出、是否关闭宽带宽模式（**Wide-bandwith mode**，该模式默认是开启的，用于缩短 **PLL** 从使能到有效频率输出之间的延时）；最后，开启 **PLLn** 的使能标志，并等待 **POSCSR** 寄存器中 **LOCKn** 标志被系统置位——**PLLn** 完成锁定并有有效频率输出。

在程序运行时，对于工作中的 **PLLn** 进行参数修改，并不需要首先将其使能关闭。当参数被改变时，**PLLn** 会自动屏蔽时钟输出，**POSCSR** 对应的 **LOCKn** 表示将被清零。直到 **PLLn** 再次完成锁定，自动设置 **LOCKn** 标志位并恢复时钟输出。

与 **PLLn** 相关的寄存器设置，可以查阅器件手册 **Power Manager** 章节的 **UserInterface** 小结。涉及到的寄存器有：**PLL0** 和 **PLL1**。其它相关的寄存器还有 **POSCSR**、**IER**、**IDR**、**IMR** 和 **IFR**。

7.3.3 同步时钟与 CPU/HSB/PBA/PBB 时钟域

同步时钟（**Synchronouse Clock**）又称系统时钟，用于驱动 **CPU** 和外设，是微处理器的脉搏。由于一个系统内同一时刻只能有一个有效的系统时钟，**CPU** 和所有外设的时钟输入都来自于这个时钟源（或者来自这个时钟源的某个分频后的结果），起到了同步设备工作和通讯的作用，因而被称为同步时钟。

在 **AVR32 UC3** 系列微处理器的内部，有一个默认的低速 **RC** 振荡器，以接近 **115KHz** 的速度运行，被称为低速时钟（**Slow Clock**）。每次复位以后，系统都会默认以低速时钟作为系统时钟。低速时钟几乎可以算作是系统的生物钟，看门狗、**PLL** 锁相环、实时时钟 **RTC** 等外设都会不同程度的利用低速时钟作为对时间的衡量。正因如此，除了 **CPU** 的静态休眠模

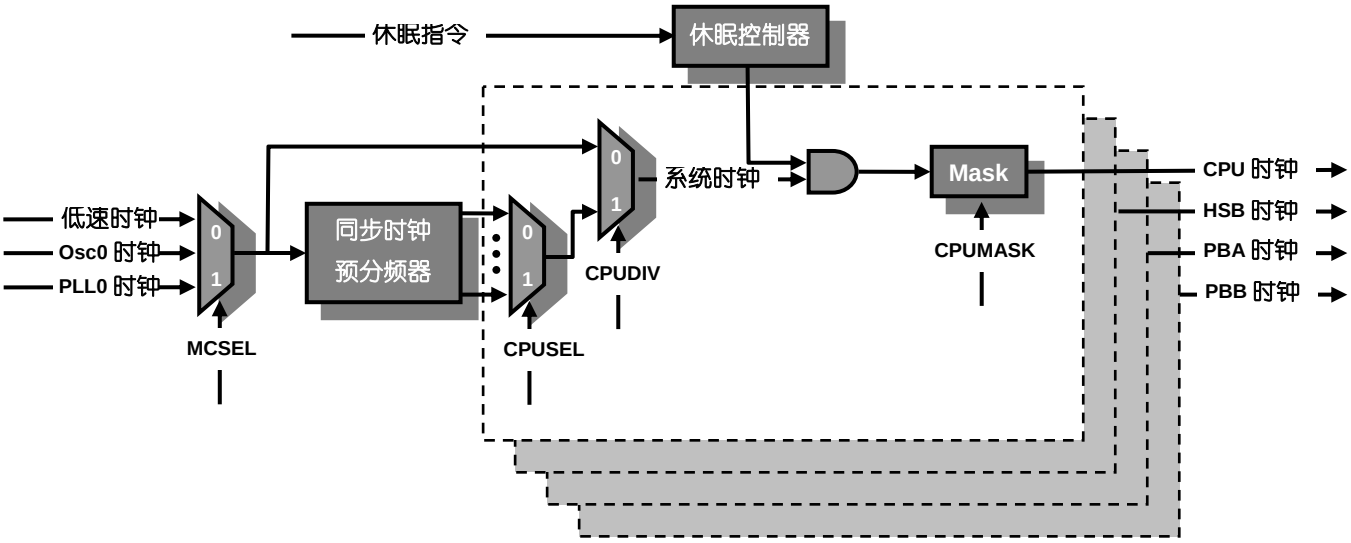


图 7.3.4 同步时钟发生器模块结构图

式以外（该模式关闭所有时钟信号甚至是看门狗），低速时钟一直保持独立的运行状态。关于它的详细参数，可以从具体芯片的数据手册上获得，这里就不再赘述。

除了默认的低速时钟以外，只有**OSC0** 和**PLL0** 的时钟输出可以作为系统时钟源（如所图 **7.3.1** 示）。在同步时钟的分配上，**AVR32 UC3** 系列将系统分为 **4** 大时钟域，分别是：**CPU**、**HSB**、**PBA**和**PBB**。四大时钟域在引入系统同步时钟时，可以采用不同的分频设置，只要满足**CPU**的频率大于等于**HSB**、**PBA**和**PBB**即可。事实上，由于设计上的原因，**AVR32 UC3** 系列的某些型号**HSB**的同步时钟设置与**CPU**相同。（相关细节请留意所使用具体芯片的器件手册。）普通情况下，每个时钟域都有一个同步时钟分频开关和具体的分频设置寄存器。通常分频开关被命名为**xxxDIV**（例如**PBADIV**），而保存具体分频数值的寄存器选项被称为**xxxSEL**（例如**PBASEL**）。当**xxxDIV**被置位时，表示开启对应时钟域对同步时钟的分频，具体的分频数值等于 $1/2^{(xxxSEL+1)}$ 倍系统时钟。注意，当**xxxDIV**被清零时，请务必将**xxxSEL**设置为“**0**”以保证硬件系统可靠运行。

对同步时钟的设置主要分为两个阶段：

- 同步时钟源的设定。主要是从低速时钟（**Slow Clock**）、**OSC0** 和 **PLL0** 中选择一个有效的时钟源。需要注意的是，如果所选择时钟源并没有被正确的初始化，或者无法输出有效时钟信号的话，将会造成死锁。只有复位可以解除这一状态。

- 四大时钟域对输入主时钟的分频设置。同步时钟会自动地被分频，产生 **8** 路固定分频的输出结果，他们分别是 **2** 分频、**4** 分频、**8** 分频……和 **256** 分频。四大时钟域可以从 **8** 路信号中选择一个，也可以使用未经分频的同步时钟。对于 **CPU** 时钟域来说，额定最大的时钟频率不能超过 **66M**，而其他时钟域也不能超过 **CPU** 时钟域的时钟。

时钟域的设置以及同步时钟源的设置可以在程序运行的时候进行而不用关闭时钟域的时钟或者让处理器挂起。在我们改变了设置以后，会有一轻微的延时，此时 **POSCSR** 寄存器的 **CKRDY** 标志会被清零，在 **CKRDY** 标志恢复置位之前不应该修改任何同步时钟的设置参数，否则系统将无法正常工作。

与同步时钟相关的寄存器设置，可以查阅器件手册 **Power Manager** 章节的 **UserInterface** 小结。涉及到的寄存器有：**MCCTRL**、**CKSEL**、**CPUMASK**、**HSBMASK**、**PBAMASK** 和 **PBBMASK**。其它相关的寄存器还有 **POSCSR**、**IER**、**IDR**、**IMR** 和 **IFR**。

7.3.4 低功耗与休眠

CPU、**HSB**、**PBA** 和 **PBB** 四大时钟域的划分，其最终目的是实现系统的最小功耗控制。（参照表 **7.3.1**）仔细观察可以发现，除去 **CPU** 时钟域外，**HSB** 上主要连接着 **FLASH** 控制器、设备总线桥 **A**、设备总线桥 **B**、**USB**、网络模块、设备 **DMA** 控制器、**EBI** 这样需要高速大数据量吞吐的总线或设备模块，可以说是系统的主动脉；设备总线 **A** 主要连接一些低速外设，或者说相对访问量较小的外设，对于 **SPI** 这种可能需要高速大数据量的操作，往往直接由 **PBA** 设备总线上的设备总线 **DMA** 控制器 **PDCA** 来完成，基本不需要 **CPU** 过多的干涉；对于设备总线 **B** 来说，主要连接着一些需要高速访问的设备模块。对于截然不同的操作速度和数据吞吐需求，四大时钟域可谓各有专攻。设想，如果采用统一的系统时钟分频，那么 **PBA** 总线与 **PBB** 总线将不得不使用相同的频率，通过降低时钟的方法来降低 **PBA** 上低速设备的功耗将成为一句空话。

表 **7.3.1** 四大时钟域可屏蔽设备一览表

Bit	CPUMASK	HSBMASK	PBAMASK	PBBMASK
0	-	FLASHC	INTC	HMATRIX
1	OCD-	PBA bridge	GPIO	USBB-
2	-	PBB bridge	PDCA	FLASHC
3	-	USBB	PM/RTC/EIC	MACB
4	-	MACB	ADC	SMC
5	-	PDCA	SPI0	SDRAMC
6	-	EBI	SPI1	-
7	-	-	TWI	-
8	-	-	USART0	-
9	-	-	USART1	-
10	-	-	USART2	-
11	-	-	USART3	-
12	-	-	PWM	-
13	-	-	SSC	-

四大时钟域的划分，实现了宏观上基于工作时钟频率的划分，在一定程度上降低了芯片功耗。而时钟域内部模块级别的时钟开关管理，则可以将不用的模块直接关掉，实现功耗控制的精打细算。

表 7.3.1 列出了一个典型的时钟域内部可屏蔽设备一览表。表头中 **xxxMASK** 表示具体时钟屏蔽寄存器的名称。当把某一模块在寄存器的对应二进制位清零时，将关断该模块的时钟供应。

从表上看，有一些模块在两个时钟域内都有屏蔽标志，这表面上看似乎隐含了某些模块的时钟可以同时来自多个时钟域的观点；但另一方面，如果两个时钟域可以拥有对系统时钟不同的分频，那么这两个时钟是如何在同一个模块里和平共处呢？观察器件手册的系统模块结构图可以发现，以 **USBB** 模块为例，连接在 **HSB** 时钟域的并不是 **USBB** 模块，而是 **USBB** 模块专用的 **DMA** 控制器，**HSB** 能控制的只是对该 **DMA** 控制器的时钟屏蔽，简单来说，如果在 **HSBMASK** 寄存器中清除了 **USBB** 对应的二进制位，仅仅表示关闭了 **USBB** 专用的 **DMA** 时钟，无法通过 **HSB** 来访问 **USBB** 模块而已，仍然可以通过 **PBB** 总线来访问 **USBB**；如果我们清除了 **PBBMASK** 寄存器中 **USBB** 的时钟屏蔽位，将彻底关断 **USBB**，但不影响 **HSB** 供应给 **USBB** 专用 **DMA** 控制器——两种操作对应不同的功耗削减，功能上的效果也大相径庭。

对于 **CPU** 时钟域来说，我们可以通过屏蔽 **OCD** 模块的时钟来防止外部通过 **OCD** 模块在运行时干预 **CPU** 的行为——除非用户直接利用 **JTAG** 驱动 **SAB** 控制器，执行 **CHIP_ERASE** 指令，实现整片擦除，或者干脆重新对芯片进行编程。这种方法与设置芯片 **Protect** 标志不同，它并不禁止下载和 **HSB** 总线的访问。

当 **CPU** 也处于空闲模式时，切断 **CPU** 时钟以达到降低功耗为目的的模式，我们称之为休眠。表 **7.3.2** 大致列举了不同深度的休眠模式下系统的运行状态。包括唤醒方式在内的详细情形请参考具体的器件手册。我们将在第四篇中详细讲解低功耗模式下的软件开发原则。

表 7.3.2 休眠模式一览表

Index	休眠模式	CPU	HSB	PBA/B GCLK	Osc0,1 PLL0,1	Osc32	RCOsc	BOD & Bandgap	电压校准
0	Idle	Stop	Run	Run	Run	Run	Run	On	全电压
1	Frozen	Stop	Stop	Run	Run	Run	Run	On	全电压
2	Standby	Stop	Stop	Stop	Run	Run	Run	On	全电压
3	Stop	Stop	Stop	Stop	Stop	Run	Run	On	低电压
4	DeepStop	Stop	Stop	Stop	Stop	Run	Run	Off	低电压
5	Static	Stop	Stop	Stop	Stop	Stop	Stop	Off	低电压

与低功耗和休眠相关的寄存器设置，可以查阅器件手册 **Power Manager** 章节的 **UserInterface** 小结。涉及到的寄存器有：**MCCTRL**、**CKSEL**、**CPUMASK**、**HSBMASK**、**PBAMASK** 和 **PBBMASK**。其它相关的寄存器还有 **POSCSR**、**IER**、**IDR**、**IMR** 和 **IFR**。

7.3.5 普通时钟

普通时钟（**Generic Clock**）是一个相对于同步时钟（**Synchronouse Clock**）的概念，它主要用于为某些有特殊时钟需求的外设提供特定频率的时钟源。例如，**USBB** 工作势需要一个 **48M** 的通讯时钟；**ABDAC** 在音频输出时需要一个精确的 **44.1Khz** 的频率。从同一个 **OSC** 或者 **PLL** 引入时钟并同时满足系统时钟和特殊外设的需求通常是不可能的，因此 **AVR32 UC3** 系列微处理器将同步时钟和普通时钟的概念区别对待，并增加了对至少两个 **OSC** 和 **PLL** 的支持，希望能够产生更多的频率组合，满足系统对时钟的需求。

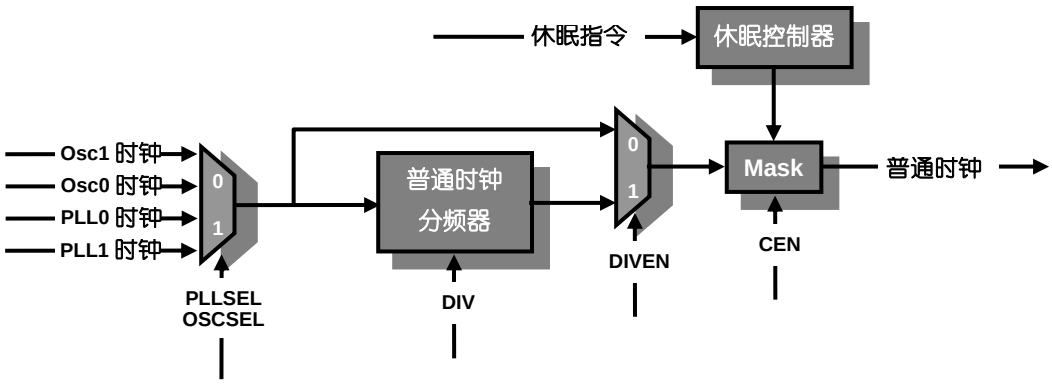


图 7.3.5 普通时钟发生器模块结构图

普通时钟的内部逻辑结构如图 7.3.5 所示。普通时钟既可以选择 **OSCn** 也可以选择 **PLLn** 作为输入时钟源，通过分频开关 **DIVEN** 和分频器 **DIV** 的处理，产生所需的普通时钟。当 **DIVEN** 为 **1** 时，输入的时钟源将被除以 **2(DIV + 1)**。同样普通时钟也有一个时钟输出屏蔽标志 **CEN**，用于管理是否向所连接的目标设备输出时钟。值得注意的是，不同型号的 **AVR32 UC3** 芯片拥有不同数量的普通时钟，而这些时钟都一一对应于一个专门的模块。表 **7.3.3** 就是一个典型的例子。从这张表上我们可以知道，该型号的芯片拥有 **6** 路独立的普通时钟。其中前 **4** 路普通时钟用于在 **GCLKn** 引脚上输出特殊的频率、**4** 号用于供给 **USBB**、**5** 号用于供给 **ABDAC**。

表 7.3.3 AVR32 UC3A0/A1 普通时钟功能分配表

普通时钟编号	功能
0	在引脚 GCLK0 上输出时钟
1	在引脚 GCLK1 上输出时钟
2	在引脚 GCLK2 上输出时钟
3	在引脚 GCLK3 上输出时钟
4	提供给 USBB
5	提供给 ABDAC

普通时钟（**Generic Clock**）的设定与同步时钟的设定类似。对每一路普通时钟的设定来说，首先需要从 **OSC0**、**OSC1**、**PLL0** 和 **PLL1** 中选择一个作为输入时钟源。接下来，可以选择是否对输入的时钟信号进行分频，以及如何分频。最后，通过设置 **CEN** 标志，开启该路普通时钟的输出。在不关闭同步时钟的情况下，可以通过清除 **CEN** 表示来暂时屏蔽该路时钟信号。当我们试图通过普通时钟通道输出特定的频率时，还需要将相关的 **GPIO** 引脚与普通时钟关联起来，具体的操作方法可以参照第三篇的 **GPIO** 章节。

在程序的运行时刻，如果我们想改变某个普通时钟的设置，必须先关闭该普通时钟（清除使能标志）。新的参数设置完成以后再设置使能标志。这与同步时钟是不同的。**一个系统内只有一同步时钟，却可以拥有多个不同的普通时钟。**同步时钟的设置与普通时钟并不冲突，**OSCn** 和 **PLLn** 都可以同时成为同步时钟和普通始终的输入时钟源。对同步时钟的设置可以通过 **Software Framework** 提供的接口函数来完成。

与普通时钟（**Generic Clock**）相关的寄存器设置，可以查阅器件手册**Power Manager**章节的**UserInterface**小结。涉及到的寄存器有：**GCCTRL**。

7.3.6 系统复位

PM 中的复位控制器（**Reset Controller**）管理着所有的复位源，主要包括两大类，软件复位（**CPU Error**、**OCD**）以及硬件复位（**External Reset**、**Power-on Reset**、**Brownout Reset**、**Watchdog Timer**）。

软件复位 **CPU Error**、**OCD** 都是由软件产生，区别是 **CPU Error** 复位产生的原因是 **CPU** 在特权模式是非法的访问了外部的寄存器，而 **OCD** 复位则是因为调试应用的需要。

硬件复位 **External Reset**、**Power-on Reset**、**Brownout Reset**、**Watchdog Timer** 不同程度都是监测芯片运行的模块，针对不同的情况进行相应的复位。**External Reset** 是最基本的复位源，有独立的复位引脚提供。掉电检测（**Brown-Out Detector**）用于监测运行中芯片 **1.8V** 内核电压，可以通过熔丝位设定不同的电压，并且可以设定为电压异常时是产生复位还是产生中断信号。上电检测（**Power-On Detector**）则用于检测确认电源上电时的电压情况，只有电压符合电压参数以及平稳要求时才能释放复位信号，**POD** 是唯一对芯片所有外设进行复位的模块。而且，**Watchdog Timer** 则是标准的程序运行监测机制。

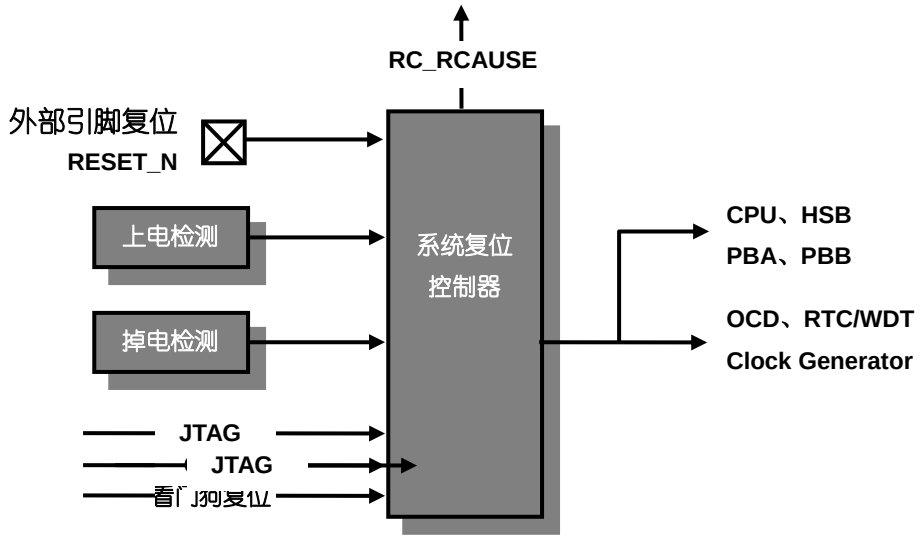


图 7.3.6 复位控制器系统框图

表 7.3.4 AVR32 UC3A0/A1 系统复位模式一览表

	Power-On Reset	External Reset	Watchdog Reset	BOD Reset	CPU Error Reset	OCD Reset
CPU/HSB/PBA/PBB (包括 PM)	Y	Y	Y	Y	Y	Y
32KHz oscillator	Y	N	N	N	N	N
RTC control register	Y	N	N	N	N	N
GPLP register	Y	N	N	N	N	N
Watchdog control register	Y	Y	N	Y	Y	Y
Voltage Calibration register	Y	N	N	N	N	N
RC Oscillator Calibration register	Y	N	N	N	N	N
BOD control register	Y	Y	N	N	N	N
Bandgap control register	Y	Y	N	N	N	N
Clock control register	Y	Y	Y	Y	Y	Y
Osc0/Osc1 and control register	Y	Y	Y	Y	Y	Y
PLL0/PLL1 and control register	Y	Y	Y	Y	Y	Y
OCD system and OCD register	Y	Y	N	Y	Y	N

AVR32.UC3.PM.API 硬件驱动

如表 7.4.1 所示，AVR32 UC3 的电源管理模块寄存器众多、操作复杂，通过直接访问寄存器的方法来设置 PM 将是一件相当痛苦的事情。本着为用户提供快速开发平台的思想，AVR32 Software Framework 以发布硬件 API 驱动函数库的方法为用户封装了很多接口函数。通过这些接口函数，我们不必理会具体的寄存器细节，就可以轻松的完成整个系统的电源管理。

本章后面的内容，我们将略过寄存器的具体操作，以用户如何使用 Software Framework 为蓝本为大家介绍电源管理模块的操作方式。其中，我们会补充一些关于使用 API 函数所必须的数据结构和常量定义。这些信息在工程中的位置以及详细参考信息的获取方法也将一一列出。

表 7.4.1 AVR32 UC3A0/A1 Power Manager 用户接口寄存器一览表

地址偏移	寄存器	名称	读写权限	复位状态
0x0000	Main Clock Control	MCCTR	读/写	0x00000000
0x0004	Clock Select	CKSEL	读/写	0x00000000
0x0008	CPU Mask	CPUMASK	读/写	0x00000003
0x000C	HSB Mask	HSBMASK	读/写	0x0000007F
0x0010	PBA Mask	PBAMASK	读/写	0x0000FFFF
0x0014	PBB Mask	PBBMASK	读/写	0x0000003F
0x0018	Reserved			
0x001C	Reserved			
0x0020	PLL0 Control	PLL0	读/写	0x00000000
0x0024	PLL1 Control	PLL1	读/写	0x00000000
0x0028	Oscillator 0 Control Register	OSCCTRL0	读/写	0x00000000
0x002C	Oscillator 1 Control Register	OSCCTRL1	读/写	0x00000000
0x0030	Oscillator 32 Control Register	OSCCTRL32	读/写	0x00000000
0x0034	Reserved			
0x0038	Reserved			
0x003C	Reserved			
0x0040	PM Interrupt Enable Register	IER	只写	0x00000000
0x0044	PM Interrupt Disable Register	IDR	只写	0x00000000
0x0048	PM Interrupt Mask Register	IMR	只读	0x00000000
0x004C	PM Interrupt Status Register	ISR	只读	0x00000000
0x0050	PM Interrupt Clear Register	ICR	只写	0x00000000
0x0054	Power and Oscillators Status Register	POSCSR	读/写	0x00000000
0x0058	Reserved			
0x005C	Reserved			
0x0060	Generic Clock Control 0	GCCTRL	读/写	0x00000000
0x0064	Generic Clock Control 1	GCCTRL	读/写	0x00000000
0x0068	Generic Clock Control 2	GCCTRL	读/写	0x00000000
0x006C	Generic Clock Control 3	GCCTRL	读/写	0x00000000
0x0070	Generic Clock Control 4	GCCTRL	读/写	0x00000000
0x0074	Generic Clock Control 5	GCCTRL	读/写	0x00000000
0x0078	Reserved			
0x007C~0x00BC	Reserved			

0x00C0	RC Oscillator Calibration Register	RCCR	读 / 写	工厂设定
0x00C4	Bandgap Calibration Register	BGCR	读 / 写	工厂设定
0x00C8	Linear Regulator Calibration Register	VREGCR	读 / 写	工厂设定
0x00CC	Reserved			
0x00D0	BOD Level Register	BOD	读 / 写	Flash 中的熔丝设定
0x00D4	Reserved			
0x00D8	Reserved			
0x00DC~0x013C	Reserved			
0x0140	Reset Cause Register	RCAUSE	只读	最新的复位源
0x0144	Reserved			
0x0148	Reserved			
0x014C~0x1FC	Reserved			
0x0200	General Purpose Low-Power register 0	GPLP0	读 / 写	0x00000000
0x0204	General Purpose Low-Power register 1	GPLP1	读 / 写	0x00000000

7.4.1 数据结构

当我们在工程中通过 **#include <avr32/io.h>** 包含了对芯片基本的输入输出支持时，实际上通过条件编译选择了工程设置中所选芯片型号的支持库，比如 **avr32/uc3a0512.h**。在这类 **uc3xxxxx.h** 文件中以 **AVR32_PM** 为名称定义了电源管理模块在芯片线性空间中的地址，并定义了很多与使用 **PM** 相关的宏，例如：

```
//uc3a0512.h
/* PM */
#define AVR32_PM_NUM                1                   //UC3A0512 中 PM 的数量

/* PM */
#define AVR32_PM_ADDRESS            0xFFFF0C00          //PM 的在地址空间中的绝对地址
#define AVR32_PM                    (*((volatile avr32_pm_t*)AVR32_PM_ADDRESS))
#define AVR32_PM_CLK_PBA                67
#define AVR32_PM_IRQ                    41
#define AVR32_PM_GCLK_ABDAC             5
#define AVR32_PM_GCLK_MSB               3
#define AVR32_PM_GCLK_NUM               6
#define AVR32_PM_GCLK_USBB              4
#define AVR32_PM_GPLP_NUM               2
……

#include "avr32/pm_231.h"
```

在上面代码片断的最后，我们注意到一个名为 **pm_231.h** 的文件被包含了进来。其实 **avr32** 目录下还有很多以 **pm_xxx.h** 为结构命名的头文件，它们分别被用来描述几类拥有不同配置的电源管理模块，并在头文件中定义了 **Software Framwork** 内部操作电源管理模块所必需的结构体和常量，例如：

```
//pm_231.h
……
//操作寄存器 MCCTRL 的位域
```

```
typedef struct avr32_pm_mcctrl_t {
    unsigned int                :28;
    unsigned int osc1en         : 1;
    unsigned int osc0en         : 1;
    unsigned int mcsel          : 2;
} avr32_pm_mcctrl_t;

//操作寄存器 CKSEL 的位域
typedef struct avr32_pm_cksel_t {
    unsigned int pbbdiv         : 1;
    unsigned int                : 4;
    unsigned int pbbsel         : 3;
    unsigned int pbadiv         : 1;
    unsigned int                : 4;
    unsigned int pbasel         : 3;
    unsigned int hsbdiv         : 1;
    unsigned int                : 4;
    unsigned int hsbsel         : 3;
    unsigned int cpudiv         : 1;
    unsigned int                : 4;
    unsigned int cpusel         : 3;
} avr32_pm_cksel_t;

……
//定义电源管理模块的寄存器接口
typedef struct avr32_pm_t {
  union {
        unsigned long                mcctrl     ;//0x0000
        avr32_pm_mcctrl_t            MCCTRL     ;
  };
  union {
        unsigned long                cksel      ;//0x0004
        avr32_pm_cksel_t             CKSEL      ;
  };
  ……
} avr32_pm_t;
……
```

由此我们知道在 **uc3xxxxx.h** 中定义的宏 **AVR32_PM** 实际上是把 **pm_xxx.h** 中定义的电源管理模块寄存器接口描述结构体 **avr32_pm_t** 强制绑定在宏 **AVR32_PM_ADDRESS** 所指定的地址上。**AVR32_PM** 实际上就是一个 **avr32_pm_t** 类型的变量，其地址固定为 **AVR32_PM_ADDRESS**。

在 **pm_xxx.h** 中，定义了很多用于操作对应寄存器的位域，例如上例中的 **avr32_pm_mcctrl_t**。在电源管理模块寄存器接口描述结构体中，它被与 **U32** 变量 **mcctrl** 绑定

在一起，被命名为 **MCCTRL**。我们可以很方便的通过这些复合接口访问任意寄存器中的任意位域，例如：

```
//将 MCCTRL 中的 OSC1 使能标志位 osc1en 置一，开启 Oscillator 0
AVR32_PM.MCCTRL. osc1en = 1;
```

即便如此，通过这种方法直接操作寄存器来完成电源管理模块的设定是不推荐的。**ATMEL** 官方一直希望 **Software Framework** 的作用就是为用户提供一个绕过硬件细节直接快速快发的平台。另一方面，通过了解 **uc3xxxxx.h** 和 **pm_xxx.h** 中的内容，观察 **AVR32 Software Framework** 关于电源管理模块的函数封装方法，对从事 **AVR32** 底层驱动开法的工程师来说还是相当有帮助的。

7.4.2 硬件驱动 API 函数

AVR32 UC3 Software Framework 为用户提供了对电源管理模块的硬件接口函数。这些函数被封装在 **Software Framework\DRIVERS\PM** 文件夹下的 **pm.c** 中；**pm.h** 提供了对这些接口函数的说明。

7.4.2.1 Oscillator 0、Oscillator 1 和 32KHz Oscillator

```
/* 该函数将 Oscillator 0 的工作模式设置为外部时钟，此时XIN引脚应该连接外部时钟的输出端，
   而 XOUT则不用特殊处理——它已经还原为普通的GPIO了。*/
/* 输入参数：&AVR32_PM */
extern void pm_enable_osc0_ext_clock(volatile avr32_pm_t *pm);
```

```
/* 该函数选择石英晶体振荡器作为OSC0的输入时钟源，系统会自动根据传递给函数的晶振频率
   选择正确的工作模式 */
/* 输入参数：&AVR32_PM，连接晶振的实际工作频率 */
extern void pm_enable_osc0_crystal(volatile avr32_pm_t *pm, unsigned int fosc0);
```

```
/* 该函数将以指定的startup time设置来使能OSC0，并等待OSC0启动完成 */
/* 输入参数：&AVR32_PM，晶体振荡器的启动时间设置，具体输入内容参照下表 */
/*
   设置值        延时的低速时钟周期        启动参考时间
     0             0                        0
     1             64                       560us
     2             128                      1.1ms
     3             2048                     18ms
     4             4096                     36ms
     5             8192                     71ms
     6             16384                    142ms
     7             Reserved                 Reserved

你也可以使用定义在pm_xxx.h中的宏来设定startup time参数
# define   AVR32_PM_OSCCTRL0_STARTUP_0_RCOSC                    0x00000000
```

```
# define   AVR32_PM_OSCCTRL0_STARTUP_128_RCOSC          0x00000002
# define   AVR32_PM_OSCCTRL0_STARTUP_16384_RCOSC        0x00000006
# define   AVR32_PM_OSCCTRL0_STARTUP_2048_RCOSC         0x00000003
# define   AVR32_PM_OSCCTRL0_STARTUP_4096_RCOSC         0x00000004
# define   AVR32_PM_OSCCTRL0_STARTUP_64_RCOSC           0x00000001
# define   AVR32_PM_OSCCTRL0_STARTUP_8192_RCOSC         0x00000005
*/
extern void pm_enable_clk0(volatile avr32_pm_t *pm, unsigned int startup);
```

```
/* 该函数将关闭OSC0的使能 */
/* 输入参数：&AVR32_PM   */
extern void pm_disable_clk0(volatile avr32_pm_t *pm);
```

```
/* 该函数将以指定的startup time设置来使能OSC0，但是并不等待OSC0启动完成 */
/* 输入参数：（与pm_enable_clk0()相同） */
extern void pm_enable_clk0_no_wait(volatile avr32_pm_t *pm, unsigned int startup);
```

```
/* 该函数用于确认当前的Oscillator 0是正常工作，时钟源是有效的。当Oscillator 0处于
   内部新参数生效的调整状态时，该函数将一直等待，直到该过程完成。一个典型的例子：
   pm_enable_clk0函数实际上是pm_enable_clk0_no_wait和
   extern void pm_wait_for_clk0_ready的整合*/
/* 输入参数：&AVR32_PM   */
extern void pm_wait_for_clk0_ready(volatile avr32_pm_t *pm);
```

以下的 OSC0 设置请参考 OSC0。

```
//Oscillator 1
extern void pm_enable_osc1_ext_clock(volatile avr32_pm_t *pm);
extern void pm_enable_osc1_crystal(volatile avr32_pm_t *pm, unsigned int fosc1);
extern void pm_enable_clk1(volatile avr32_pm_t *pm, unsigned int startup);
extern void pm_disable_clk1(volatile avr32_pm_t *pm);
extern void pm_enable_clk1_no_wait(volatile avr32_pm_t *pm, unsigned int startup);
extern void pm_wait_for_clk1_ready(volatile avr32_pm_t *pm);
```

```
/* 该函数用于选择外部时钟作为OSC32的输入时钟源 */
/* 输入参数：&AVR32_PM   */
extern void pm_enable_osc32_ext_clock(volatile avr32_pm_t *pm);
```

```
/* 该函数用于选择32KHz高精度石英晶体振荡器作为输入时钟源 */
/* 输入参数：&AVR32_PM   */
extern void pm_enable_osc32_crystal(volatile avr32_pm_t *pm);
```

```
/* 该函数将以指定的startup time设置来使能OSC32，并等待OSC32启动完成 */
/* 输入参数：&AVR32_PM，晶体振荡器的启动时间设置，具体输入内容参照下表 */
/*
```

设置值	延时的低速时钟周期	启动参考时间
0	0	0
1	128	1.1 ms
2	8192	72.3 ms
3	16384	143 ms
4	65536	570 ms
5	131072	1.1 s
6	262144	2.3 s
7	Reserved	Reserved

你也可以使用定义在pm_xxx.h中的宏来设定startup time参数

```
# define   AVR32_PM_OSCCTRL32_STARTUP_0_RCOSC                    0x00000000
# define   AVR32_PM_OSCCTRL32_STARTUP_128_RCOSC                  0x00000001
# define   AVR32_PM_OSCCTRL32_STARTUP_131072_RCOSC               0x00000005
# define   AVR32_PM_OSCCTRL32_STARTUP_16384_RCOSC                0x00000003
# define   AVR32_PM_OSCCTRL32_STARTUP_262144_RCOSC               0x00000006
# define   AVR32_PM_OSCCTRL32_STARTUP_524288_RCOSC               0x00000007
# define   AVR32_PM_OSCCTRL32_STARTUP_65536_RCOSC                0x00000004
# define   AVR32_PM_OSCCTRL32_STARTUP_8192_RCOSC                 0x00000002
*/
extern void pm_enable_clk32(volatile avr32_pm_t *pm, unsigned int startup);
```

```
/* 该函数将关闭OSC32的使能 */
/* 输入参数：&AVR32_PM   */
extern void pm_disable_clk32(volatile avr32_pm_t *pm);
```

```
/* 该函数将以指定的startup time设置来使能OSC32，但是并不等待OSC32启动完成 */
/* 输入参数：（与pm_enable_clk32()相同） */
extern void pm_enable_clk32_no_wait(volatile avr32_pm_t *pm, unsigned int startup);
```

```
/* /* 该函数用于确认当前的32KHz Oscillator是正常工作，时钟源是有效的。当OSC32处于
   内部新参数生效的调整状态时，该函数将一直等待，直到该过程完成。*/
/* 输入参数：&AVR32_PM   */
extern void pm_wait_for_clk32_ready(volatile avr32_pm_t *pm);
```

7.4.2.2 PLLn

```
/* 该函数设置并启动指定编号PLL的时钟参数 */
/* 输入参数：&AVR32_PM，PLL编号0或1，倍频因子，
            分频因子，选择的OSCn时钟源0或1,锁定延迟 */
extern void pm_pll_setup
            (
                volatile avr32_pm_t *pm,     //&AVR32_PM
                unsigned int pll,            //0或1表示PLL1或者PLL0
                unsigned int mul,            //乘法因子，取值范围 4~15
```

```
                unsigned int div,           //分频因子，取值范围 0~3
                unsigned int osc,           //0或1表示OSC0或者OSC1
                unsigned int lockcount      //在设置LOCKn标志之前额外延时的低速时钟周期
            );
```

```
/* 该函数用来设定指定编号PLL的各项属性 */
/* 输入参数：&AVR32_PM，PLL编号，fvco时钟范围0或1，
            是否将fvco二分频后输出，是否关闭宽带宽模式 */
extern void pm_pll_set_option
            (
                volatile avr32_pm_t *pm,    //&AVR32_PM
                unsigned int   pll,         //0或1表示PLL1或者PLL0
                unsigned int   pll_freq,    // fvco的频率范围：
                                            //1 对应80M~180MHz    0 对应160M~240MHz
                unsigned int   pll_div2,    //0或1，1表示将fvco二分频后作为PLLn时钟输出
                unsigned int   pll_wbwdisable   //0或1，1表示关闭宽带宽模式，一般取值为0
            );
```

```
/* 该函数用于获取指定编号PLL的各项寄存器设定，对于返回的U32整数，你可以通过强制邦定为
   寄存器位域avr32_pm_pll_t的方法来获得具体的寄存器设置，例如：

//pm_231.h中关于avr32_pm_pll_t的定义
typedef struct avr32_pm_pll_t {
    unsigned int plltest            : 1;
    unsigned int plliotesten        : 1;
    unsigned int pllcount           : 6;
    unsigned int                    : 4;
    unsigned int pllmul             : 4;
    unsigned int                    : 4;
    unsigned int plldiv             : 4;
    unsigned int pllbpl             : 1;
    unsigned int                    : 2;
    unsigned int pllopt             : 3;
    unsigned int pllosc             : 1;
    unsigned int pllen              : 1;
} avr32_pm_pll_t;

   //代码范例：获取PLL0的寄存器设置
   U32 wPLLRegister = pm_pll_get_option(&AVR32_PM,0);
   avr32_pm_pll_t *ppllREG = (avr32_pm_pll_t *)&wPLLRegister;
   //通过ppllREG->pllmul可以获取当前的倍频因子
*/
/* 输入参数：&AVR32_PM，PLL编号 */
extern unsigned int pm_pll_get_option(volatile avr32_pm_t *pm, unsigned int pll);
```

```
/* 该函数用于使能指定编号的PLL，并不等待PLLn产生有效的输出 */
/* 输入参数：&AVR32_PM，PLL编号  */
extern void pm_pll_enable(volatile avr32_pm_t *pm, unsigned int pll);
```

```
/* 该函数用于关闭指定编号的PLL */
/* 输入参数：&AVR32_PM，PLL编号  */
extern void pm_pll_disable(volatile avr32_pm_t *pm, unsigned int pll);
```

```
/* 该函数用于等待PLL0产生稳定的时钟输出。PLL0可以在不关闭使能的情况下直接修改参数，
   每次修改参数以后，PLL0会自动屏蔽时钟输出，应该通过该函数等待PLL0新的参数生效 */
/* 输入参数：&AVR32_PM */
extern void pm_wait_for_pll0_locked(volatile avr32_pm_t *pm);
```

```
/* 该函数用于等待PLL1产生稳定的时钟输出。PLL1可以在不关闭使能的情况下直接修改参数，
   每次修改参数以后，PLL1会自动屏蔽时钟输出，应该通过该函数等待PLL1新的参数生效 */
/* 输入参数：&AVR32_PM */
extern void pm_wait_for_pll1_locked(volatile avr32_pm_t *pm);
```

7.4.2.3 系统时钟和 CPU/HSB/PBA/PBB 时钟域

```
/* 该函数用于设定HSB/PBA/PBB在引入同步时钟时的分频设置，需要注意的是，对于某些型号的
   芯片来说HSB和CPU设置总是相同的，因此只需要设置HSB时钟分频即可，相关内容可以查阅对
   应的器件手册 */
/* 输入参数：&AVR32_PM,PBA的分频开关，PBA的分频选择，PBB的分频开关，PBB的分频选择，
             HSB（CPU）的分频开关，HSB的分频选择 */
extern void pm_cksel
            (
                volatile avr32_pm_t *pm,     //&AVR32_PM
                unsigned int pbadiv,         //PBA总线分频开关，0表示不分频
                unsigned int pbasel,         //PBA的分频，当PBADIV为0时，该项也应为0
                unsigned int pbbdiv,         //PBB总线分频开关，0表示不分频
                unsigned int pbbsel,         //PBB的分频，当PBBDIV为0时，该项也应为0
                unsigned int hsbdiv,         //HSB总线分频开关，0表示不分频
                unsigned int hsbsel          //HSB的分频，当HSBDIV为0时，该项也应为0
            );
```

```
/* 该函数将选择指定的时钟源作为当前的系统时钟 */
/* 输入参数：&AVR32_PM，时钟源编号 */
/*
   时钟源编号只能是以下常数之一
        AVR32_PM_MCSEL_SLOW        芯片自带的默认115K RC振荡器
        AVR32_PM_MCSEL_OSC0        选择Oscillator 0作为系统主时钟
        AVR32_PM_MCSEL_PLL0        选择PLL0作为系统主时钟
```

```
*/
extern void pm_switch_to_clock(volatile avr32_pm_t *pm, unsigned long clock);
```

```
/* 该函数直接选择OSC0作为系统时钟源，函数默认使用石英晶体振荡器，并根据传入的晶振频率和
   启动时间startup time参数自动设置OSC0 */
/* 输入参数：&AVR32_PM，OSC0上连接晶振的实际频率，晶振的启动时间 */
extern void pm_switch_to_osc0
                    (
                        volatile avr32_pm_t *pm,      //&AVR32_PM
                        unsigned int fosc0,           //OSC0连接晶振的频率
                        unsigned int startup          //startup time设置
                    );
```

7.4.2.4 普通时钟

```
/* 该函数用于设置指定通道编号的普通时钟，选择输入时钟源，分频开关以及分频参数等 */
/* 输入参数：&AVR32_PM, 普通时钟通道编号，OSC或PLL选择，OSC0/OSC1或者PLL0/PLL1选择
             分频开关，分频设置 */
extern void pm_gc_setup
                (
                    volatile avr32_pm_t *pm,      //&AVR32_PM
                    unsigned int gc,              //普通时钟通道编号，请参阅具体器件手册
                    unsigned int osc_or_pll,      //0表示OSC，1表示PLL
                    unsigned int pll_osc,         //当选择OSC时，0表示OSC0、1表示OSC1
                                                  //当选择PLL时，0表示PLL0、1表示PLL1
                    unsigned int diven,           //分频开关 0表示不分频
                    unsigned int div              //当DIVEN为1时，输入的时钟将被处以
                                                  //2 * (DIV + 1)
                );
```

```
/* 该函数将开启指定通道普通时钟的使能标志 */
/* 输入参数：&AVR32_PM，普通时钟的通道编号 */
extern void pm_gc_enable(volatile avr32_pm_t *pm, unsigned int gc);
```

```
/* 该函数将关闭指定编号的普通时钟通道 */
/* 输入参数：&AVR32_PM，普通时钟的通道编号 */
extern void pm_gc_disable(volatile avr32_pm_t *pm, unsigned int gc);
```

7.4.2.5 其它

```
/* 这是一个参数宏，使用在线汇编的方法封装了一个接口，用于设定当前系统的休眠模式 */
/* 输入参数：只能是以下宏所代表的常量之一 */
/*  ----------------------------------------------------------------------------------------------
     宏                                                    说明
```

```
    AVR32_PM_SMODE_IDLE                 Idle
    AVR32_PM_SMODE_FROZEN               Frozen
    AVR32_PM_SMODE_STANDBY              Standby
    AVR32_PM_SMODE_STOP                 Stop
    AVR32_PM_SMODE_SHUTDOWN             Shutdown (DeepStop)
    AVR32_PM_SMODE_STATIC               Static
*/  ------------------------------------------------------------------------------------------
#define SLEEP(mode)    {__asm__ __volatile__ ("sleep "STRINGZ(mode));}
```

```
/* 该函数将开启BOD的中断使能，当电压低于掉电门限时，将触发该中断 */
/* 输入参数：&AVR32_PM */
extern void pm_bod_enable_irq(volatile avr32_pm_t *pm);
```

```
/* 该函数将关闭BOD的中断功能 */
/* 输入参数：&AVR32_PM */
extern void pm_bod_disable_irq(volatile avr32_pm_t *pm);
```

```
/* 该函数用来清除BOD中断标志 */
/* 输入参数：&AVR32_PM */
extern void pm_bod_clear_irq(volatile avr32_pm_t *pm);
```

```
/* 该函数用于读取BOD当前的检测结果，返回1表示触发了BOD掉电保护 */
/* 输入参数：&AVR32_PM */
/* 输出参数：0或1 */
extern unsigned long pm_bod_get_irq_status(volatile avr32_pm_t *pm);
```

```
/* 该函数用来了解当前是否开启了BOD中断使能，返回1表示中断被使能 */
/* 输入参数：&AVR32_PM */
/* 输出参数：0或1 */
extern unsigned long pm_bod_get_irq_enable_bit(volatile avr32_pm_t *pm);
```

```
/* 该函数用来了解当前系统设定的BOD掉电保护门限的等级 */
/* 输入参数：&AVR32_PM */
/* 输出参数：当前的BOD掉电门限等级，具体含义请参考器件手册 */
extern unsigned long pm_bod_get_level(volatile avr32_pm_t *pm);
```

```
/* 该函数用来读取制定的GPLP寄存器 */
/* 输入参数：&AVR32_PM，GPLP寄存器的编号 */
/* 输出参数：读取到的GPLP寄存器的值 */
extern unsigned long pm_read_gplp(volatile avr32_pm_t *pm, unsigned long gplp);
```

```
/* 该函数用于向制定编号的GPLP寄存器写入指定的值 */
/* 输入参数：&AVR32_PM，GPLP寄存器的编号，要写入的值 */
extern void pm_write_gplp(volatile avr32_pm_t *pm, unsigned long gplp, unsigned long value);
```

```
/* 该函数假设OSC0上连接了一个12MHz的晶振，在此基础上将自动使用PLL1产生一个普通时钟，
   提供给USB控制器。 */
extern void pm_configure_usb_clock(void);
```

AVR32.UC3.PM.Examples 范例

7.5.1 设置 PLL 至 48Mhz

［需求与分析］

本范例将通过对 **PM** 的设置，使芯片的主频运行至 **48MHz**，并设定 **PB19** 作为 **GCLK** 输出 **PLL1** 锁相环所产生的 **48MHz** 波形，通过延时翻转 **LED1** 的状态。与 **OpenUC3** 相关的宏定义，都可以在 **SOFTWARE_FRAMEWORK\BOARDS\OpenUC3\OpenUC3.h** 中找到，如：

```
/* SOFTWARE_FRAMEWORK\BOARDS\OpenUC3\OpenUC3.h */
……
/* 定义LED引脚 */
# define LED1_PIN                    AVR32_PIN_PB27

/* 定义GCLK输出管教 */
# define EXAMPLE_GCLK_ID             0
# define EXAMPLE_GCLK_PIN            AVR32_PM_GCLK_0_1_PIN
# define EXAMPLE_GCLK_FUNCTION       AVR32_PM_GCLK_0_1_FUNCTION
……
```

［相关设计资料］

核心代码如下：

```
// Include Files
#include "board.h"
#include "pm.h"
#include "gpio.h"
#include "flashc.h"

void local_start_pll0(volatile avr32_pm_t* pm)
{
    //直接选择 OSC0 上的晶振作为系统主时钟，FOSC0 和 OSC0_STARTUP 都在 BSP 中定义
    pm_switch_to_osc0(pm, FOSC0, OSC0_STARTUP);

    //设置锁相环
    pm_pll_setup
        (
            pm,
```

```
            0,      //  设置  PLL0
            7,      // 8 倍频
            1,      //  不分频
            0,      //使用 OSC0 作为 PLL0 的输入时钟源
            16
        );

    //设置 PLL0 的各项属性
    pm_pll_set_option
        (
            pm,
            0,      //设置 PLL0
            1,      //选择 fvco 的输出时钟范围  80M ~ 180MHz
            1,      //将 fvco 二分频以后作为 PLL 的时钟输出
            0       //保持宽带宽模式开启，将获得较快的 PLL 锁定时间
        );

    //开启 PLL0 的使能
    pm_pll_enable(pm,0);

    //等待 PLL0 完成锁定
    pm_wait_for_pll0_locked(pm) ;

    //设置普通时钟
    pm_gc_setup
        (
            pm,
            EXAMPLE_GCLK_ID,            //设置 EXAMPLE_GCLK_ID 所指定的通道
            1,                          //以 PLL 作为输入时钟源
            0,                          //以 PLL0 作为输入时钟源
            0,                          //对输入的时钟不进行分频操作
            0
        );

    //使能 EAMPLE_GCLK_ID 所指定的普通时钟通道
    pm_gc_enable(pm, EXAMPLE_GCLK_ID);

    //设置引脚 EXAMPLE_GCLK_PIN 使用指定的设备功能——普通时钟输出
    gpio_enable_module_pin(EXAMPLE_GCLK_PIN, EXAMPLE_GCLK_FUNCTION);

    //设置(CPU)HSB/PBA/PBB 的总线分频  */
    pm_cksel
        (
            pm,
```

```
            1,      //PBA 开启总线分频
            0,      //默认为 2 分频，即 PBA 工作在 48M/2 = 24MHz 的频率下
            0,      //PBB 关闭总线分频
            0,      //写 0 以保持系统正常工作
            0,      //HSB 关闭总线分频
            0       //写 0 以保持系统正常工作
        );
    //当 HSB(CPU)的同步时钟高于 30M 时，必须增加这句话，我们将在 FLASHC 章节中
    //详细为您解释这个问题
    flashc_set_wait_state(1);

    //选择 PLL0 输出的时钟频率作为系统同步时钟源
    pm_switch_to_clock(pm, AVR32_PM_MCSEL_PLL0);
}

//软件延时函数
static void software_delay(void)
{
  volatile int i;
  for (i=0; i<1000000; i++);
}

int main(void)
{
    //获取 Power Manager 模块的接口指针
    volatile avr32_pm_t* pm = &AVR32_PM;

    //调用前面编写的函数，用以设定当前的系统时钟，详细设置参考 local_start_pll0()函数
    local_start_pll0(pm);

    while(1)
    {
        //将宏 LED1_PIN 所指定的引脚电平取反，LED1_PIN 在 BSP 中定义
        gpio_tgl_gpio_pin(LED1_PIN);
        software_delay();
    }
}
```

［演示效果］

正确下载程序到 **OpenUC3**，我们会看到评估版上的 **LED1** 在闪烁，如果读者有示波器可以观察 **PB19** 所产生的 **48MHz** 波形情况。

AVR32.UC3.PM.Reference 相关参考信息

7.5 如何获取进一步的参考信息

作为一本 **AVR32 UC3** 工程技术指南，我们不可能涵盖官方文档以及驱动函数的所有信息。除了本章提炼的信息以外，更详细的内容可以通过以下的渠道获得：

- 关于**PM**寄存器的详细信息以及相互间的制约关系，可以参阅官方器件手册的相关章节：**13. Power Manager (PM)**。本书参考的章节版本为**REV 2.0.0.1**。
- **ATMEL**官方提供的驱动函数库中，**include/avr32**件夹下的**pm_100.h**、**pm_200.h**、**pm_210**、**pm_230.h**和**pm_231.h**以及**Driver/PM**文件夹下的**pm.h**和**pm.c**将会为我们带来更多的信息。
- 在官方器件手册 **8. I/O Line Consideration**中有关于外部中断引脚**RESET_N**的详细介绍。
- 关于低功耗模式下的软件设计，可以参考官方应用文档 **AVR32739: AVR32 UC3 Low power software design**