501 的省电管理

V1.0

Publication Release Date: May. 2009

Table of Contents

1. 前言

NUC501 是以 ARM7TDMI 为核心的 32 位单片微机，提供高达 108MHz 的运作频率，满足高速运算需求的工作。

然而在一般简单控制或系统负载较低时，CPU 并不需要执行在最高的运行频率下即可应付所需要的计算量，这时候，便可以利用系统频率的控制来降低频率，以达到省电的目的。

另一种情况是 CPU 因为等待下一笔工作或只需要久久工作一次时，就可以先将 CPU 进入 Idle 或 Power Down 模式等到一定的时间之后或是有特定的事件发生时，才醒过来进行相关事件的处理，同时，如果系统内没有用到的系统模块，也可以将其关闭以节省不必要的耗电。

接下来的章节将会介绍各种不同的省电方法的原理，以及提供相关的范例说明。

2. 开关系统模块

2.1. 为什么要关闭系统模块

在 NUC501 中，为了省电的需求，当某个系统模块不使用时，可以将其输入频率关闭，使其进入关闭的模式，依此来达到省电的效果。

在 501 中，几乎所有的系统模块都可以各别被关闭，这些模块包括了：

APU, SPIM, USB, ADC, SPIMS, I2C, PWM, UART, RTC, Watch Dog Timer and TIMER.

各模块的相关耗电如下表所示：

IP Enable/ Disable	Current Consumption(mA)
APU	1.24
SPIM	1.8
USB	1.17
ADC	0.48
SPIMS	0.40
I2C	0.24
PWM	0.47
UART	0.07
RTC	0.22
Watch Dog Timer	0.11
TIMER	0.11

Note: 以上数据由 3.3Volt 的情况下测得，由于环境及制程因素，可能会有些许变动。

2.2. 如何开关各模块

501 藉由关闭各模块频率的方式来达到关闭各模块的目的，所以要关闭模块就必须设定相关的频率控制缓存器，分别为 AHBCLK Register 与 APBCLK Register，这两个缓存器的功能说明如下表所示：

- **AHB 各模块频率控制缓存器 (AHBCLK)**

在 AHBCLK 缓存器内的各位，被用来作为 AMBA clock, AHB Bus 上的模块以及 APB Bus 的频率开关。

Register	Address	R/W	Description	Reset Value
AHBCLK	0xB1000200 + 04	R/W	AHB 模块频率控制缓存器	0x0000_0083

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							APU_CK_EN
7	6	5	4	3	2	1	0
SPIM_CK_EN	USBD_CK_EN	Reserved				APB_CK_EN	CPU_CK_EN

Bits	Descriptions	
[31:9]	Reserved	Reserved
[8]	APU_CK_EN	APU 模块的频率控制 0 = 关闭频率 1 = 开启频率
[7]	SPIM_CK_EN	SPIM 模块的频率控制(SPIM0 & SPIM1) 0 = 关闭频率 1 = 开启频率
[6]	USBD_CK_EN	USB 模块的频率控制 0 = 关闭频率 1 = 开启频率
[5:2]	Reserved	Reserved
[1]	APB_CK_EN	APB Bus 的频率控制 0 = 关闭频率 1 = 开启频率
[0]	CPU_CK_EN	CPU 的频率控制 0 = 关闭频率 1 = 开启频率

- **APB 各模块频率控制缓存器（APBCLK）**

APBCLK 缓存器内的各个位被用来控制 APB Bus 上各模块的频率。

Register	Address	R/W	Description	Reset Value
APBCLK	0xB1000200 + 08	R/W	APB Devices Clock Enable Control Register	0x0000_0007

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved						ADC_CK_EN	SPIMS_CK_EN
7	6	5	4	3	2	1	0
Reserved	I2C_CK_EN	PWM_CK_EN	UART1_CK_EN	UART0_CK_EN	RTC_CK_EN	WD_CK_EN	TIMER_CK_EN

Bits	Descriptions	
[31:10]	Reserved	Reserved
[9]	ADC_CK_EN	Analog-Digital-Converter (ADC)模块的频率控制 0 = 关闭频率 1 = 开启频率
[8]	SPIMS_CK_EN	SPIMS 模块的频率控制 0 = 关闭频率 1 = 开启频率
[7]	Reserved	Reserved
[6]	I2C_CK_EN	I2C 模块的频率控制 0 = 关闭频率 1 = 开启频率
[5]	PWM_CK_EN	PWM 模块的频率控制 0 = 关闭频率 1 = 开启频率
[4]	UART1_CK_EN	UART1 模块的频率控制 0 = 关闭频率 1 = 开启频率
[3]	UART0_CK_EN	UART0 模块的频率控制 0 = 关闭频率 1 = 开启频率
[2]	RTC_CK_EN	Real-Time-Clock (RTC) 模块的 APB 接口频率控制。这个位只能用来控制 RTC 模块与 APB 连接接口的频率，而不能控制由外部所提供的 32.768kHz crystal 频率。 0 = 关闭频率 1 = 开启频率
[1]	WD_CK_EN	Watch Dog Timer (WDT) 模块的频率控制 0 = 关闭频率 1 = 开启频率
[0]	TIMER_CK_EN	Timer 模块的频率控制。 0 = 关闭频率 1 = 开启频率

想要控制特定模块的开关，只需要找到相对应的控制位，并将其设置为 0 或 1 即可。一般可以透过 501BSP 内的 DrvSYS Driver 来完成。例如要关闭 UART1 模块时，可以使用下面的代码来完成：

DrvSYS_SetIPClock(E_DRVSYS_UART1_CLK, 0);

如果想要直接通过缓存器的设定来开关各个模态的频率，则可以使用底层的 I/O 存取，下面的代码便演示了如何直接关闭 UART1 的频率：

```
outp32(APBCLK, inp32(APBCLK) & (~(1 << 4)));
```

上述的代码，直接使用了 outp32/inp32 直接对 APBCLK 缓存器作设置，因为在这个例子中我们只想要将 UART1 关闭，所以需要先将 APBCLK 的设置读回来，改动要设定的位后，再写回 APBCLK 缓存器中，以免误关到别的模块。

3. 系统频率设定

NUC501 本身的耗电量和它的运行频率有很有的关系，如果运行的频率高，则耗电高，反之则比较省电，但同时 CPU 就可能无法负荷大量的运算。因此如果要同时兼顾大量运算跟省电的话，就必须能够根据实际上的需求来调整 CPU 的工作频率，以求达到最佳的运作效率。

NUC501 内建了 PLL，能使用 2MHz ~ 20MHz 的 Crystal 产生系统所需的频率，由 PLL 所产生的频率再经过适当除频，即可作为 CPU 的工作频率，因此 501 对于工作频率的设定，有相当大的弹性，除了快到可以运行到 108MHz，必要时甚至可以直接使用外部的 Crystal 频率作为系统的频率，例如使用 12MHz 的 Crystal 等等，使 CPU 运行在极低的频率下。

想要设定 501 的工作频率，可直接在程序里调用 DrvSYS 库里函数来完成，主要的相关函数如下：

- *DrvSYS_Open(UINT32 u32ExtClockKHz, UINT32 u32PllClockKHz)*
- *DrvSYS_SetCPUClock(UINT32 u32CpuClockKHz)*

DrvSYS_Open 是用来设定 PLL 输出频率的函数，它的输入参数有两个，第一个是要输入系统外部所使用的 Crystal 震荡频率，另一个参数则指定了 PLL 的输出频率，另外还必须注意这两个参数是以 1000Hz 为单位。例如当系统所使用的 Crystal 震荡频率是 12MHz，而需要 PLL 输出 216MHz 的话，那可调用如下的程序来进行设定：

```
DrvSYS_Open(12000, 216000);
```

其中第一个参数 12000 表示现在系统外部所使用的 Crystal 震荡频率是 12MHz，第二个参数表示要将 PLL 的输出设定为 216MHz。

当 PLL 的输出频率决定了之后，接下在就是要决定 CPU 的频率，这个设定可通过呼叫 DrvSYS_SetCPUClock( ) 来完成，此函数的参数同样是以 1000Hz 为单位，而且必须注意由于硬件限制，CPU 的频率至少要小于或等于 PLL 频率的二分之一，也就是说如果 CPU 要使用 108Mhz 的频率，那么 PLL 最少就要有 216MHz 的输出频率才行。

```
DrvSYS_SetCPUClock(108000);
```

需要特别注意的是，调用这些频率的设定函数并不一定真的能将相关频率设定成想要的样子，主要是因为硬件有它一定的限制，因此如果有需要确认目前系统实际上的运作频率的话，可通过 DrvSYS_GetPLLClock()及 DrvSYS_GetCPUClock()来取得。

501 允许用户在程序中重复呼叫 DrvSYS_Open/DrvSYS_SetCPUClock 来重新设定运作频率，不过由于重新设定 PLL 输出频率需要有一定的程序，将会造成一定程度的延迟，所以如果有动态改变系统运作频率的需要时，建议利用改变 CPU 频率的方式来完成即可，如此可将切换运作频率的延迟减到最低。

4. IDLE 省电模式

501 除了可利用设定最适合的工作频率来达到省电的目的外，如果在系统完全不需要工作时，还可以将 CPU 及大部分的硬件关闭，以达到最大的省电效果，这样子搭配关闭 CPU 及大部分硬件的模式，我们称之为省电模式，其中又包括了 Idle 省电模式及 Power Down 省电模式两种，下面我们将先就 Idle 省电模式进行说明。

4.1. 什么是 IDLE 省电模式

藉由关闭大部分硬件频率以达到最大省电效率的模式主要有两种，一种是 Idle 省电模式，另一种是 Power Down 省电模式，这两种模式最大的不同点是，当系统进入 Idle 省电模式下时，任何的中断事件(interrupt)，都可以重新唤醒 CPU，好让系统可以处理新进的事件，但是如果系统是处于 Power Down 省电模式的话，就只有少数特定的中断事件能够唤醒系统。

4.2. 如何进入 IDLE 省电模式

由于所谓的 Idle 模式，实际上是把 CPU 的频率关闭，使得在没有工作时，让 CPU 能够处在最省电的模式，而且因为在这个模式下，只是将 CPU 的频率关闭，一旦有任何中断事件发生，马上就可以打开 CPU 的频率来处理相关的事件，因此有不会造成处理事件的延迟，又可以达到省电的效果，是兼顾速度与省电的一种模式。

要进入 Idle 模式，可以透过呼叫 DrvSYS 库的 DrvSYS_EnableIdle()来达成，如下面的程序所示：

```
DrvSYS_EnableIdle();
```

一旦进入 Idle 模式，CPU 将立刻停止运作，因此在 DrvSYS_EnableIdle()函数之后得程序都必须等到 CPU 被唤醒后才会被执行。

一旦进入 Idle 省电模式，501 的耗电量将降到约 360uA。

4.3. 如何离开 IDLE 省电模式

进入 Idle 省电模式之后，如果要唤醒 CPU，回到一般的工作模式，就必须先产生中断讯号，这中断讯号可以是 501 中的任何模块所产生的中断，也可以透有外部中断的方式来唤醒 CPU，而由 Idle 省电模式唤醒 CPU 后，再进入中断事件处理函数的延迟时间约 1.7us。要注意的是，当决定 CPU 要由某个中断来唤醒时，就必须在进入 Idle 省电模式前，将该中断设定完成，才能够使其产生中断事件来唤醒 CPU。至于中断事件的设定，可根据不同的模块，利用 501BSP 中相对应的驱动库来完成。

5. Power Down 省电模式

5.1. 什么是 Power Down 省电模式

Power Down 是 501 最省电的一种模式，因为一旦进入 Power Down 省电模式，Crystal 频率会被关闭，整颗 501 呈现静止的状态，这时的耗电量将减到最小的程度，而在此模式下，也只有少数的特定中断事件能够唤醒 501，使其恢复工作状态。

5.2. 如何进入 Power Down 省电模式

要进入 Power Down 省电模式，可以透过呼叫 DrvSYS 库的 DrvSYS_EnablePowerDown()来达成，如下面的程序所示：

DrvSYS_EnablePowerDown();

DrvSYS_EanblePowerDown()将会设定 501 的 Power Down 控制缓存器，501 将在缓存器设定完成后再延迟 256 个外部 Crystal 频率才真正将外部 Crystal 输入关闭，此时才算是真正进入 Power Down 省电模式，因此在程序中要注意 DrvSYS_EnablePowerDown()之后的程序是有可能会在进入 Power Down 省电模式之前被执行到的，通常可以在呼叫 DrvSYS_EnablePowerDown()后，加入一段延迟的循环，来确保 CPU 在进入 Power Down 省电模式前，不会再进行额外的动作。

由于在 Power Down 省电模式下，整个频率都停止了，所以数字电路的部分耗电量会最小，但因为 PLL 是属于模拟模块，因此如果没有特别将 PLL 关闭的话，即使关闭它的输入频率，它仍会运作并输出无法预期的频率讯号，因而消耗约 200uA 的电流，所以要真正将 501 Power Down，就必须先把 PLL 关闭，而关闭 PLL 前必须先将系统使用的频率切换成使用外部 Crystal，才能让 CPU 进行接下来真正进入 Power Down 省电模式的动作。

另一点要注意的是 501 在进入 Power Down 之后，由于是系统频率关闭，所以几乎所有的模块及 GPIO 都会保持 Power Down 以前的状态，但是其中 GPIOA[11:8]将被强迫切换成输入状态且 GPIO[11:8]内部的 Pull Up 电阻此时也会无作用。

在进入 Power Down 之后，如果有 I/O 脚位是再输入状态，但却没有真正的信号输入，可能会造成浮接，也就是输入的信号无法确定是 0 或 1，这将会造成漏电的问题，因此在 Power Down 时，如果有任何脚位是处于输入状态的话，我们必须确保其不会有浮接的状况，如果确认相关脚位会有浮接的状况，可以通过加入 Pull Up 电阻来解决。如果是 GPIO 脚位，那我们可以使用内部的 Pull Up 电路即可，但是因为 GPIOA[11:8]是由内部电路强迫其切换成输入模式并使得内部 Pull Up 电阻失效，所以只能使用外部的 Pull Up 或 Pull Down 电路来避免浮接情况发生。

综合以上各个要注意的事项，我们可以简单的将整个进入 Power Down 省电模式的完整程序叙述如下：

1. 先确认进 Power Down 后，不会有浮接的输入 I/O，包括 GPIOA[11:8]也要注意
2. 将系统频率设定成外部 Crystal 频率
3. 设定 MPLLCON 缓存器，将 PLL Power Down
4. 呼叫 DrvSYS_EnablePowerDown()，开始进入 Power Down

下面列出上述流程的代码以供参考：

```
/* The GPIO status should be checked before power down to avoid I/O leakage caused by floating pin. */
    /* outp32(GPIOA_PUEN, 0xFFFF); */
    /* outp32(GPIOB_PUEN, 0xFFFF); */
    /* outp32(GPIOC_PUEN, 0xFFFF); */

    /* Use external clock */
    DrvSYS_SetSystemClockSource(E_DRVSYS_SYS_EXT);

    /* Power down the PLL */
    outp32(MPLLCON, inp32(MPLLCON) | (1 << 16));

    /* Disable crystal to enter power down */
    DrvSYS_EnablePowerDown();

    /* Wait for more than 256 external crystal cycles for entering power down */
    delay = 0x1000;
    while(delay--);
```

一旦进入 Power Down 省电模式，501 的耗电量将降到约 25uA，如果进入 Power Down 省电模式下，却量到比 25uA 高很多的耗电量，就很有可能是 PLL 没有关闭或是有 I/O 漏电的问题产生。

5.3. 如何脱离 Power Down 省电模式

一旦 501 进入 Power Down 省电模式，由于大部分的逻辑电路都进入停止状态，要想唤醒 CPU，和 Idle 省电模式下，只要有任何中断即可唤醒 CPU 不同，不过相同的是，两者都必须要在进入省电模式前，先设定好将来要用来唤醒 CPU 的模块，这包括其中断或唤醒功能，这些为了唤醒 CPU 的准备都完成之后，才能进入省电模式，否则就没有任何方式能够再唤醒 CPU 了。

由于所谓的 Power Down 省电模式，实际上是把 Crystal 的频率关闭，让整个 501 呈现静止的状态，而且因为在这个模式下，Crystal 也被关闭了，一旦有 Wakeup 中断事件发生，需要等到 Crystal 稳定下来后，CPU 才能继续运作，所以由 Power Down 进入一般的工作模式，会需要延迟一点时间，这一段延迟时间的长短是可以设定的，需根据 Crystal 的稳定性来调整，由实际测量的结果显示，当 Wakeup 事件发生，到执行中断处理函数，约需要 360us，如果为了 Crystal 的稳定，另外加上了延迟设定，则必须根据延迟设定值再加上 360us。

这里所谓的延迟时间设定值，是由 PWRCON[8:23]来决定，最小值为 0，相当于关闭延迟设定(PWRCON[1])时的延迟时间，延迟设定值是以外部 Crystal 频率乘上 256 为单位，如果外部以 12MHz

Crystal 为频率输入源的话，而且 PWRCON[8:23]设置为 1，则其延迟设定约为 1/12MHz * 256 约等于 21.33us，再加上系统原本就有延迟 360us，系统由 Wakeup 事件发生到进入中断处理函数的延迟将变成约 381.33us。

下表是 PWRCON 缓存器的详细说明：

Register	Address	R/W	Description	Reset Value
PWRCON	0xB1000200 + 00	R/W	Power Down 控制缓存器	0x00FF_FF03

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Pre-Scale[15:8]							
15	14	13	12	11	10	9	8
Pre-Scale[7:0]							
7	6	5	4	3	2	1	0
Reserved				INT_EN	INTSTS	XIN_CTL	XTAL_EN

Bits	Descriptions	
[31:24]	Reserved	Reserved
[23:8]	Pre-Scale	Pre-Scale counter 当系统由 Power Down(Crystal 关闭)状态下，重新进入正常工作模式时，用来设定等待外部 Crystal 稳定的延迟值，等频率稳定后才让其进入系统，此设定值的单位为外部 Crystal 频率乘上 256。
[7:4]	Reserved	Reserved
[3]	INTSTS	Power Down 中断状态 0 = 一般状态 1 = 系统由 Power Down 恢复正常运作，这可能是由 GPIO、UART 或 RTC 来唤醒系统的 要清除本状态，要对 INTSTS 写 1
[2]	INT_EN	Power Down 中断控制开关 0 = 关闭 Power Down 中断 1 = 开启 Power Down 中断 当打开 Power Down 中断开关之后，每当 XTAL_EN 由 0 变 1 时，就会产生中断。
[1]	XIN_CTL	此位是用来控制 Pre-Scale 的设定值是否生效 1 = 启用 Pre-Scale 设定值 0 = 关闭 Pre-Scale 设定值
[0]	XTAL_EN	Crystal Oscillator (Power Down) 控制开关 1: 启用 Crystal oscillation(一般工作模式) 0: 关闭 Crystal oscillation(Power Down 省电模式)

要脱离 Power Down 省电模式，并不是单纯的打开模块的中断功能即可，而是要特别去打开该模块的唤醒功能，目前有提供由 Power Down 省电模式唤醒功能的模块有：

- 外部讯号 (GPIO interrupt)
- UART
- RTC

，除此之外，如果在 Power Down 前，我们有先将 PLL 关闭，那么唤醒系统之后的第一件事，通常就是先将 PLL 打开，以确保接下来的程序运作正常。下面我们将一一说明各个唤醒方式如何实现。

5.3.1. 由外部讯号唤醒

要由外部讯号唤醒 CPU，可以使用外部中断的方式，设定方式跟一般外部中断相同，但有两点要特别注意，一个是需要另外打开该中断的唤醒功能，也就是要呼叫 DrvGPIO_EnableWakeupInt()来打开该中断的唤醒功能，另外就是中断的触发方式要选择上下缘都产生中断，也就是呼叫 DrvGPIO_EnableInt()时要设定成 IO_BOTH_EDGE。

下面是以使用 GPIOA[3]来由 Power Down 省电模式唤醒 CPU 的范例：

```
/* Enable Key A interrupt to wakeup the CPU */
DrvGPIO_Open(GPIOA, 3, IO_INPUT, IO_NO_PULL_UP, IO_LOWDRV);
DrvGPIO_EnableInt(GPIOA, 3, IO_INT0, IO_BOTH_EDGE, (DRVGPIO_CALLBACK)IoHandler, 0);
DrvGPIO_EnableWakeupInt(IO_INT0);

/* Set I bit of ARM */
DrvAIC_SetCPSR(AIC_ENABLE_IRQ);
```

其中 IoHandler()是外部中断时的处理函数，其至少要包含将中断旗标清除的动作，否则会造成一直发出中断讯号，且一直唤醒 CPU 导致无法再进入 Power Down 省电模式，简单的 IoHandler()范例如下：

```
VOID IoHandler(UINT32 u32UserData)
{
    UINT32 *pu32Reg;

    pu32Reg = DrvGPIO_GetIntStatus();
    pu32Reg[0] = (1 << 3); /* Clear the interrupt flag */

}
```

请注意因为我们在设定外部中断时，使用了上下缘触发，因此这个 IoHandler()将在外部有任何变化时都会被呼叫到。

5.3.2. 由 UART 讯号唤醒

UART 模块是利用 Modem 的中断事件讯号，来对 Power Down 下的系统做唤醒的动作，在 501 中，这个讯号就是 CTS。

因此如过要利用 UART 来唤醒 CPU，首先必须要让 UART 的 CTS/RTC 脚位有连接到外部，这可透过呼叫 DrvGPIO_SetPadReg1()来完成，例如我们要使用 UART0 来唤醒 CPU，就要使用以下的程序先将相关的 GPIO 切换成 UART 加上 CTR/RTS 的功能：

```
DrvGPIO_SetPadReg1(FUNC_UART0, 3);
```

I/O 脚位设定好了之后，接下来就是要打开 UART 的 Modem 中断事件，然后再启动由 Modem 中断事件来唤醒 CPU 的功能，其相关程序范例下：

```
/* Enable the modem interrupt and install its callback */
DrvUART_EnableInt(UART_PORT0, DRVUART_MOSINT, (PFN_DRVUART_CALLBACK *)UartHandler);

/* Enable UART0 CTS of modem interrupt to wakeup the CPU */
outp32(UA_IER, inp32(UA_IER) | (1 << 7));
```

除了设置好 Modem 的中断外，为了要打开 UART 的 Wakeup 功能，我们必须将 UA_IER 缓存器的 bit 7 设置为 1，如此当 Modem 中断发生时，才能唤醒系统。

请注意上述的范例省略了 UART 的初始化程序，而相关的代码可以在 UART 的 Driver Sample 中找到。

范例中所使用的 UartHandler 则是用来处理 UART 中断的函数，用户可以利用它来清除 UART 中断旗标或做其他功能的处理。

依上述的说明，设定好了 UART 之后，便可以利用 CTS 讯号的变化来由 Power Down 省电模式来唤醒 CPU 了。

5.3.3. 由 RTC 讯号唤醒

进入 Power Down 省电模式之后， 可以利用 RTC 闹钟(Alarm)功能来唤醒系统，由于 RTC 的运作频率为 32.768kHz，跟系统的 Crystal 频率是不一样的，所以即使进入 Power Down，RTC 仍可以正常工作，此时我们就可以利用 Alarm 功能，在特定的时间唤醒系统，当然也可以透过设定好下一次 Alarm 的时间，达到固定周期性的唤醒系统，以进行必要的工作。

下面的程序是一个周期性 Alarm 的范例，搭配之前提到进入 Power Down 省电模式的方法，即可达到固定时间唤醒，然后再进入 Power Down 的效果，在下面程序中，我们先初始化 RTC 之后，便进入无限循环中，等待 Alarm 事件的发生，在这无限循环内，可以加入 Alarm 时所要处理的工作，并在工作完成之后，再进入 Power Down 省电模式，并等待下一次的 Alarm 来唤醒系统。

```
S_DRVRTC_TIME_DATA_T g_sCurTime;

/* Time Setting */
g_sCurTime.u32Year = 2009;
g_sCurTime.u32cMonth = 1;
g_sCurTime.u32cDay = 19;
g_sCurTime.u32cHour = 13;
g_sCurTime.u32cMinute = 20;
g_sCurTime.u32cSecond = 0;
g_sCurTime.u32cDayOfWeek = DRVRTC_MONDAY;
g_sCurTime.u8cClockDisplay = DRVRTC_CLOCK_24;

DrvRTC_Init();
```

```
    /* Initialization the RTC timer */
    if(DrvRTC_Open(&g_sCurTime) !=E_SUCCESS)
            DrvSIO_printf("Open Fail!!\n");

    /* Trigger the Alarm start */
    RtcHandler();

    DrvAIC_SetCPSR(AIC_ENABLE_IRQ);

    while(1)
    {
            /* The code to handle the task after Alarm and enter power down mode again. */
    }
```

上述范例中的 RtcHandler 内容可参考下面的程序：

```
  VOID RtcHandler(VOID)
  {


    /* PLL should work before we can print message */

    /* Set PLL to normal mode */
    outp32(MPLLCON, inp32(MPLLCON) & (~(1 << 16)));

    /* Use PLL clock */
    DrvSYS_SetSystemClockSource(E_DRVSYS_SYS_PLL);


    /* Get the currnet time */
    DrvRTC_Read(DRVRTC_CURRENT_TIME, &g_sCurTime);

    /* Set Alarm call back function */
    g_sCurTime.pfnAlarmCallBack = RtcHandler;

    DrvSIO_printf("Wakeup by RTC Alarm. Current Time:%d/%02d/%02d %02d:%02d:%02d\n",
    g_sCurTime.u32Year,g_sCurTime.u32cMonth,g_sCurTime.u32cDay,g_sCurTime.u32cHour,g_sCurTime.u32cMi
  nute,g_sCurTime.u32cSecond);

    /* The alarm time setting: every 3 seconds */
    g_sCurTime.u32cSecond = (g_sCurTime.u32cSecond + 3);
    if( g_sCurTime.u32cSecond >= 60)
    {
          g_sCurTime.u32cSecond -= 60;
          g_sCurTime.u32cMinute ++;
    }

    /* Set the alarm time (Install the call back function and enable the alarm interrupt)*/
    DrvRTC_Write(DRVRTC_ALARM_TIME,&g_sCurTime);


  }
```

RtcHandler()主要的功能是读出目前的 RTC 时间，并在该时间加上 3 秒钟，来表示三秒后要再发一次 Alarm 事件，用以唤醒系统。请注意因为 RtcHandler()有使用 DrvSIO_printf()来透过 UART 输出一些提示讯息，由于 UART 需要 PLL 频率来产生 baud rate， 为了确保讯息打印没问题，程序中会在一进入 RtcHandler 便将 PLL 打开，使讯息的打印能够正常工作。

6. 总结

在本文件中，我们介绍了 501 所提供的各种省电方式，包括了关闭没用到的模块、调整 CPU 运作的频率、Idle 省电模式与 Power Down 省电模式，用户可以根据自身的应用，选择适合的方式，甚至组合不同的省电方法以达到最佳的省电效率。在 501 BSP 也提供了相关的范例程序，以供用户参考，此范例可在 501BSP\NuvotonPlatform_ADS\Sample\Diagnostic\Smpl_PowerManagement 目录下找到。

7. Revision History

Version	Date	Description
V1.0	July. 17, 2009	● Created

Important Notice

Nuvoton products are not designed, intended, authorized or warranted for use as components in equipment or systems intended for surgical implantation, atomic energy control instruments, aircraft or spacecraft instruments, transportation instruments, traffic signal instruments, combustion control instruments, or for any other applications intended to support or sustain life. Furthermore, Nuvoton products are not intended for applications whereby failure could result or lead to personal injury, death or severe property or environmental damage.

Nuvoton customers using or selling these products for such applications do so at their own risk and agree to fully indemnify Nuvoton for any damages resulting from their improper use or sales.