AN2602

应用笔记

UM8G006A 主频 16M 改 24M

注意事项

版本：V1.0.0

UNICMICRO

广芯微电子

广芯微电子（广州）股份有限公司

http://www.unicmicro.com/

条款协议

本文档的所有部分，其著作产权归广芯微电子（广州）股份有限公司（以下简称广芯微电子）所有，未经广芯微电子授权许可，任何个人及组织不得复制、转载、仿制本文档的全部或部分组件。本文档没有任何形式的担保、立场表达或其他暗示，若有任何因本文档或其中提及的产品所有资讯所引起的直接或间接损失，广芯微电子及所属员工恕不为其担保任何责任。除此以外，本文档所提到的产品规格及资讯仅供参考，内容亦会随时更新，恕不另行通知。

1. 本文档中所记载的关于电路、软件和其他相关信息仅用于说明半导体产品的操作和应用实例。用户如在设备设计中应用本文档中的电路、软件和相关信息，请自行负责。对于用户或第三方因使用上述电路、软件或信息而遭受的任何损失，广芯微电子不承担任何责任。
2. 在准备本文档所记载的信息的过程中，广芯微电子已尽量做到合理注意，但是，广芯微电子并不保证这些信息都是准确无误的。用户因本文档中所记载的信息的错误或遗漏而遭受的任何损失，广芯微电子不承担任何责任。
3. 对于因使用本文档中的广芯微电子产品或技术信息而造成的侵权行为或因此而侵犯第三方的专利、版权或其他知识产权的行为，广芯微电子不承担任何责任。本文档所记载的内容不应视为对广芯微电子或其他人所有的专利、版权或其他知识产权作出任何明示、默示或其它方式的许可及授权。
4. 使用本文档中记载的广芯微电子产品时，应在广芯微电子指定的范围内，特别是在最大额定值、电源工作电压范围、热辐射特性、安装条件以及其他产品特性的范围内使用。对于在上述指定范围之外使用广芯微电子产品而产生的故障或损失，广芯微电子不承担任何责任。
5. 虽然广芯微电子一直致力于提高广芯微电子产品的质量和可靠性，但是，半导体产品有其自身的具体特性，如一定的故障发生率以及在某些使用条件下会发生故障等。此外，广芯微电子产品均未进行防辐射设计。所以请采取安全保护措施，以避免当广芯微电子产品在发生故障而造成火灾时导致人身事故、伤害或损害的事故。例如进行软硬件安全设计（包括但不限于冗余设计、防火控制以及故障预防等）、适当的老化处理或其他适当的措施等。

目录

1 升级概述

UM800YA 芯片主频由 16M 提升至 24M 后，相关寄存器名称保持不变；系统时钟配置寄存器在不分频配置下，RCH 主频同步更新为 24M，即：

```
SYSDIV = (0x0 << 0);                    //不分频
```

图 1-1：主频不分频代码片段

主频升级后将直接影响程序执行时序及关联 RCH 的 IP，需对指定系统文件、驱动文件及外设配置进行针对性修改，以保证芯片功能正常运行。

2 主频升级影响范围

2.1 程序执行时间

主频提升后，程序执行效率提升，以下内容的时间特性发生变化:

1. 简易时延函数。
2. 中断服务函数执行时间。
3. 常规的代码运行时间。

2.2 关联 RCH 的外设 IP

所有基于 RCH 时钟的外设均受影响，需重新配置相关参数，涉及外设包括: EFC、PWM、Beeper、UART0/1/2/3、SPI、GTIMER、I2C 和 ADC。

3 文件修改

3.1 system_um800xx.c：时钟分频配置修订

为适配 24M 主频，需修订时钟分频范围：

```
    }

    switch (system_clk_hz)
    {
        case 24000000:
            SYSDIV = (0x0 << 0);                    //不分频
            system_core_clock = system_clk_hz;
            break;

        case  12000000:
            SYSDIV = (0x1 << 0);                    //2分频
            system_core_clock = system_clk_hz;
            break;

        case  6000000:
            SYSDIV = (0x2 << 0);                    //4分频
            system_core_clock = system_clk_hz;
            break;

        default:
            SYSDIV = (0x2 << 0);                    //默认为4分频
            system_core_clock = crystal_clk / 4;
            break;
    }

    CLKCON &= ~(0x1 << 0);                  //RCH为系统时钟
}
```

图 3-1：时钟分频配置修订

注：若用户从未修改过系统文件，可以直接用提供的 system_um800xx.c 文件进行替换。

3.2 eflash.c：EFC 控制器 OUS 参数修订

EFC 控制器 OUS 参数需根据 24M 主频重新设定，以保证 EFlash 在主动读写中功能正常。

```
void eflash_init(uint32_t system_clk_hz)
{

    if (system_clk_hz < 2000000)
    {
        return;
    }

    switch (system_clk_hz)    //OUS设定值需要根据主频的不同进行配置。
    {
        case 24000000:
            OUS = 23;
            break;

        case  12000000:
             OUS = 11;
            break;

        case  6000000:
            OUS = 5;
            break;

        default:
            OUS = 5;
            break;
    }

    OPSET |= (0 << 3);                    //RD_WAIT = 0

    OINTUS = 0x3f;                        //清除所有中断标志位
    OINTEN = 0x00;                        //关闭非外设所有中断功能

}
```

图 3-2：24M 主频对应的 OUS 值设定

注：若用户从未修改过驱动文件，可以直接用提供的 eflash.c 文件进行替换。

4 受影响 IP 修改

4.1 PWM

主频提升至 24M 后，PWM 输出频率同步变快，需要相应调整 pwm0/1_init 函数的参数：

1. 若要保持与 16M 情况下相同输出频率：需减小 cycle 值。
2. 若要保持原有占空比：duty 值需与 cycle 值同步调整。

```
{
    pwm1_init(cycle1, duty1, HIGH);
    pwm1_irq_init(PWM_IRQ_DISABLE,pwm1_irq_handler);
    pwm1_start();
```

```
/*******************************************************************************
 * Function     : pwm1_init
 * Description  : pwm1_init
 * Input        : uint16_t cycle 周期（周期时长计算 = 系统时钟/cycle）
 *                uint16_t duty  占空比
 *                uint8_t  level HIGH:占空比期间输出高电平；LOW:占空比期间输出低电平
 * Output       : none
 * Return       : none
 *******************************************************************************/
void pwm1_init(uint16_t cycle, uint16_t duty, uint8_t level)
{
    REG_P11_CFG = 0x03;

    if (level == LOW)
    {
        PWM1CON |= (1 << 6);        //占空比期间输出低电平
    }
    else
    {
        PWM1CON &= ~(1 << 6);       // 占空比期间输出高电平
    }

    PWM1DL = duty & 0xFF;           // 占空比低8位
    PWM1DH = (duty >> 8) & 0xFF;    // 占空比高8位

    PWM1PL = cycle & 0xFF;          // 周期低8位
    PWM1PH = (cycle >> 8) & 0xFF;   // 周期高8位

    PWM1CON |= (1 << 7);            // 使能PWM1模块
}

/*******************************************************************************
```

图 4-1：PWM 配置修改

以下给出一个输出 1M 时钟，不同占空比的参数参考：

```
#include "config.h"

volatile uint16_t duty0 = 20;   //占空比 = duty/cycle
volatile uint16_t cycle0 = 24;  //pwm频率=时钟频率/cycle

volatile uint16_t duty1 = 12;   //占空比 = duty/cycle
volatile uint16_t cycle1 = 24;  //pwm频率=时钟频率/cycle

volatile uint16_t duty2 = 4;    //占空比 = duty/cycle
volatile uint16_t cycle2 = 24;  //pwm频率=时钟频率/cycle
```

频率1M，占空比5/6

频率1M，占空比1/2

频率1M，占空比1/6

图 4-2：1M 情况下不同占空比

4.2 BEEPER

主频提升后，Beeper 原默认配置的 1kHZ、2kHZ、4kHZ 输出频率失效，实际输出为 1.5KHz、3KHz、6KHz。

若需输出上述频率，需通过 GTIMER 或 PWM 生成。

BEEPSEL	BEEP 输出频率控制: 00/11：1kHz 01：2kHz 10：4kHz

图 4-3：BEEPSEL 输出频率配置

4.3 UART0/1/2/3

主频提升后，波特率寄存器 SxREL 数值发生变化，实际输出波特率精度提升，24M 主频下各波特率对应 SxREL 值如下表:

表 4-1：UART0/1 波特率

24000000				
目标波特率		SxREL	实际波特率	误差
115200	1010.979167	1011	115385	0.16%
57600	997.958333	998	57692	0.16%
38400	984.937500	985	38462	0.16%
19200	945.875000	946	19231	0.16%
9600	867.750000	868	9615	0.16%
4800	711.500000	712	4808	0.16%
2400	399.000000	399	2400	0.00%

表 4-2：UART2/3 波特率

UART2/3		实际BPR设定值					
	BPR	十进制	16进制	BPR_H	BPR_L	实际波特率	波特率误差
115200	208.333333	208	D0	0	0xD0	115384.62	0.160256%
57600	416.666667	417	1A1	1	0xA1	57553.96	-0.079936%
38400	625.000000	625	271	2	0x71	38400.00	0.000000%
19200	1250.000000	1250	4E2	4	0xE2	19200.00	0.000000%
9600	2500.000000	2500	9C4	9	0xC4	9600.00	0.000000%
4800	5000.000000	5000	1388	0x13	0x88	4800.00	0.000000%
2400	10000.000000	10000	2710	0x27	0x10	2400.00	0.000000%

注：

1. 若使用 SDK 原厂 uart.c 驱动：无需修改，原厂驱动通过公式自动推导计算 SxREL 值，适配不同主频。

2. 若手动填写固定波特率寄存器值：需按上表更新 SxREL 配置值。

```
 ******************************************************************************/
static void uart0_set_baud_rate(uint32_t clk_hz, uint32_t baud_rate)
{
    uint32_t temp;
    uint8_t MSbyte, LSbyte;

    temp = baud_rate * 16;
    temp = (clk_hz + (temp / 2)) / temp;
    temp = 1024 - temp ;

    MSbyte = temp >> 8;
    LSbyte = temp;

    S0RELH = MSbyte & 0x3;
    S0RELL = LSbyte;
}
```

图 4-4：串口驱动配置波特率参考片段

4.4 SPI

SPI 通信频率也会随主频同步提升，如需要重新配置分频参数，即重新配置 REG_SPI_SPCR1 寄存器的 5~7 位（BR [2:0]）：

```
 * Output       : none
 * Return       : none
 ******************************************************************************/
void spi_master_init(uint8_t work_mode, uint8_t spi_baudrate_psc)
{
    REG_P1_IE |= (1 << 3);                          //P13 开启输入
    REG_P1_IE |= (1 << 4);                          //P14 开启输入
    REG_P1_IE |= (1 << 5);                          //P15 开启输入

    PRESET0 |= (1 << 3);                          //SPI复位释放
    PCLK0   |= (1 << 3);                          //SPI模块时钟使能

    REG_SPI_SPCR1 = 0x00;
    REG_SPI_SPCR2 = 0x00;
    REG_SPI_SPCR3 = 0xF0;

    REG_SPI_SPCR1 |= (1 << 2);                    //master模式
    REG_SPI_SPCR1 &= ~(1 << 3);                   //先发送MSB
    REG_SPI_SPCR1 |= (spi_baudrate_psc << 5);     //波特率分频
    REG_SPI_SPCR2 |= (1 << 4);                    //软件控制SSN

    switch (work_mode)
    {
    case WORK_MODE_0:
        REG_SPI_SPCR1 &= ~(1 << 0);             //在SCK第一个边沿采样
        REG_SPI_SPCR1 &= ~(1 << 1);             //SCK空闲时低电平
        break;

    case WORK_MODE_1:
        REG_SPI_SPCR1 |= (1 << 0);              //在SCK第二个边沿采样
        REG_SPI_SPCR1 &= ~(1 << 1);             //SCK空闲时低电平
        break;

    case WORK_MODE_2:
        REG_SPI_SPCR1 &= ~(1 << 0);             //在SCK第一个边沿采样
        REG_SPI_SPCR1 |= (1 << 1);              //SCK空闲时高电平
        break;

    case WORK_MODE_3:
        REG_SPI_SPCR1 |= (1 << 0);              //在SCK第二个边沿采样
        REG_SPI_SPCR1 |= (1 << 1);              //SCK空闲时高电平
        break;
```

图 4-5：分频参数配置代码

位编号	位符号	说明
7-5	BR[2:0]	Master 模式波特率配置位: 000：$f_{PCLK}/2$ 001：$f_{PCLK}/4$ 010：$f_{PCLK}/8$ 011：$f_{PCLK}/16$ 100：$f_{PCLK}/32$ 101：$f_{PCLK}/64$ 110：$f_{PCLK}/128$ 111：$f_{PCLK}/256$ 当通信正在进行的时候，不能修改这些位。

图 4-6：Master 模式波特率配置

注：由于 16M 与 24M 并非简单的倍频关系，所以无法 100%还原到原 16M 下的 SPI 速率，需要根据实际情况进行功能适配。

4.5 GTIMER

若使用 GTIMER 生成 PWM，则需根据实际情况修订以下寄存器值：REG_GTIM0_PSC0、REG_GTIM0_ARR0 和 REG_GTIM0_CCR0，如下图所示：

```
 * Return       : none
 ******************************************************************************/
void gtimer0_pwm_init(uint16_t arr, uint16_t psc, uint16_t ccr)
{
    PCLK1 |= (1 << 3);                                          //开Gtim0时钟使能
    PRESET1 |= (1 << 3);                                        //GTim0正常工作

    REG_P13_CFG = 0x06;                                         //P13 CH
    REG_P15_CFG = 0x04;                                         //P15 CH
//  REG_P25_CFG = 0x04;                                          //P25 CH

    REG_P14_CFG = 0x05;                                         //P14 CHN
    REG_P20_CFG = 0x06;                                         //P20 CHN
//  REG_P23_CFG = 0x05;                                          //P23 CHN

    REG_GTIM0_CR0 |= (1 << 6);                                  //使能auto-reload

    REG_GTIM0_PSC1 = (psc >> 8) & 0xFF;                         //高8位 FCK_CNT=FCK_PSC/(PSC[15:0]+1);  FCK_PSC = APBCLK
    REG_GTIM0_PSC0 = psc & 0xFF;                                //低8位 FCK_CNT=FCK_PSC/(PSC[15:0]+1);  FCK_PSC = APBCLK

    REG_GTIM0_ARR1 = ((arr - 1) >> 8) & 0xFF;                   //高8位 自动装载值
    REG_GTIM0_ARR0 = (arr - 1) & 0xFF;                          //低8位 自动装载值

    REG_GTIM0_ARRN1 = ((arr - 1) >> 8) & 0xFF;                  //高8位 自动装载值
    REG_GTIM0_ARRN0 = (arr - 1) & 0xFF;                         //低8位 自动装载值

    REG_GTIM0_CCR1 = ((ccr) >> 8) & 0xFF;                       //高8位 比较寄存器
    REG_GTIM0_CCR0 = (ccr) & 0xFF;                              //低8位 比较寄存器

    REG_GTIM0_CCRN1 = ((ccr) >> 8) & 0xFF;                      //高8位 比较寄存器
    REG_GTIM0_CCRN0 = (ccr) & 0xFF;                             //低8位 比较寄存器

    REG_GTIM0_CR1 |= (1 << 0);                                  //互补PWM和原PWM反相位

    REG_GTIM0_CR1 |= (1 << 1);                                  //避免死区功能使能

    REG_GTIM0_EGR = 0x1;                                        //产生UE事件，将PSC的值立即载入shadow寄存器
    REG_GTIM0_SR |= (1 << 0);                                   //清除UE产生的中断，否则会直接进入中断服务函数

    REG_GTIM0_CR2 |= (1 << 2);                                  //输出总使能

    REG_GTIM0_CCER |= (1 << 0);                                 //使能输出


}
```

图 4-7：GTIMER 相关寄存器值修订代码片段

注：如果用户是直接使用原厂 SDK 中的接口修改 gtimer0_pwm_init 中的传入参数即可。

以下给出一个输出 1M 时钟的 PWM 波参数配置：

```
/*****************************************************************************
 * function    : gtimer0_pwm_test
 * Description: gtimer0_pwm_test GTIMER0 PWM输出函数
 * input : none
 * return: none
 *****************************************************************************/
void gtimer0_pwm_test(void)
{
    gtimer0_pwm_init(24,1-1,12);    //1MHz PWM输出,占空比为50%

    gtimer0_irq_init(GTIMER_IRQ_DISABLE,GTIMER0_CCIE_IRQ,gtimer0_ccie_func);        //配置CCIE中断

    gtimer0_start();        //启动gtimer0计数

}
```

图 4-8：配置 1M 输出代码片段

图 4-9：实际输出的 1M 波形

4.6 I2C

I2C 传输速率由系统时钟分频决定，主频提升后需重新计算分频值，保证 I2C 通信速率符合设计要求。将下图中的计算值修改为系统主频时钟即可：

```
 * Return       : none
 ******************************************************************************/
void i2c_master_init(uint32_t i2c_speed)
{
    REG_P0_IE |= (1 << 4);                          //P04 开启输入
    REG_P1_IE |= (1 << 0);                          //P10 开启输入

    PCLK0   |= (1 << 7);                //打开I2C模块时钟
    PRESET0 |= (1 << 7);                //释放I2C复位

    P0PU &= ~(1 << 4);                  //P04上拉
    P1PU &= ~(1 << 0);                  //P10上拉

    REG_P04_CFG = 0x03;                 //I2C_SDA
    REG_P10_CFG = 0x04;                 //I2C_SCL

    REG_I2C_CLK_DIV = 16000000 / (4 * i2c_speed) - 1;   //I2C速率 = (fSYSCLK)/(4*(CLK_DIV+1))
    REG_I2C_CR1 |= (1 << 5);            //使能开漏模式
    REG_I2C_CR0 |= MEN;                 //I2C使能
}

/******************************************************************************
```

图 4-10：配置 REG_I2C_CLK_DIV 寄存器代码示例

14.3.2 I2C_CLK_DIV 时钟分频寄存器

CC01H	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
I2C_CLK_DIV	I2C_CLK_DIV							
读/写	读/写							
复位值	0							

位编号	位符号	说明
7-0	I2C_CLK_DIV	SCL 分频值，通过配置该寄存器设置 I2C 的传输速率。 f_{SCL} =（F 系统时钟）/（4*（DIV 寄存器值+1））。

图 4-11：I2C_CLK_DIV 寄存器说明

图 4-12：实际输出的 400K 波形

注：公式中修改为 24000000 时，400K 的通信时钟正常输出（要外接合适的上拉电阻）。

4.7 ADC

ADC 采样率由系统时钟分频配置决定，主频提升后，采样频率有所提升。如要调整 ADC 采样时钟，配置 clksource_div 即可。

```
 * return         : none
 *******************************************************************************/
void adc_clk_config(uint8_t clksource, uint8_t vrefsource, uint16_t clksource_div, uint8_t newstate)
{
    if(newstate == ADC_ENABLE)
    {
        PCLK0 |= (1<<4);                                        //ADC时钟使能
        PRESET0 |= (1<<4);                                      //ADC复位释放

        ADCGCR0 = (ADCGCR0 & (~(1<<6))) | (clksource<<6);       //ADC时钟源选择

        ADCCDR0 = 0;                                            //清零CLKDIV
        ADCCDR1 = 0;
        ADCCDR0 = (0x00ff & clksource_div);                     //ADC内部时钟分频倍数
        ADCCDR1 = (0xff00 & clksource_div)>>8;

        ADCVREF = (ADCVREF & (~(1<<0)))|(vrefsource<<0);        //电压参考源选择
    }
    else
    {
        PCLK0 &= ~(1<<4);                                       //ADC时钟关闭
        PRESET0 &= ~(1<<4);                                     //ADC复位
    }
}
```

图 4-13：ADC 内部时钟分频倍数修改代码示例

在与 16M 相同的分频比的情况下，24M 主频的采样结果表现正常，与 16M 表现几乎一样。

	2分频	转换值		4分频	转换值		8分频	转换值
max	1679	1.356		1652	1.335		1640	1.325
min	1551	1.253		1578	1.275		1551	1.253
ave	1608	1.299		1605	1.296		1606	1.297
	1611	1.301		1604	1.296		1627	1.314
	1605	1.297		1616	1.306		1603	1.295
	1619	1.308		1592	1.286		1612	1.302
	1611	1.301		1608	1.299		1604	1.296
	1620	1.309		1609	1.300		1588	1.283
	1613	1.303		1644	1.328		1592	1.286

图 4-14：部分 ADC 实际测试数据

5 版本修订

版本	日期	描述
V1.0.0	2026.03.13	初始版