

MSP430F552X 中文手册及例程

一、先写一篇开头：这样快速闯入 MSP430学习过程

进入各个电子产品公司的网站，招聘里面嵌入式占据了大半工程师职位。广义的嵌入式无非几种：传统的什么51单片机、MSP430称做嵌入式微控制器；ARM 是嵌入式微处理器；当然还有 DSP；FPGA。我们现在就不说别的，就说 MSP430单片机，多数想学 MSP430的童鞋，对89C51内核系列的单片机是很熟悉的，为了加深对 MSP430 系列单片机的认识吗，迅速闯入 MSP430学习过程，就必须彻底了解 MSP430单片机，我们不妨将51单片机和 MSP430两者进行一下比较。

第一点，51内核单片机是8 位单片机。其指令是采用的被称为“ CISC ”的复杂指令集，共具有111 条指令。而 MSP430 单片机是16 位的单片机，采用了精简指令集（ RISC ）结构，只有简洁的27 条指令，大量的指令则是模拟指令，众多的寄存器以及片内数据存储器都可参加多种运算。这些内核指令均为单周期指令，功能强，运行的速度快。

第二点，MCU 主要分为两种工作模式：待机与执行。51内核单片机正常情况下消耗的电流为 mA 级，在掉电状态下，其耗电电流仍约为3mA 左右；即使在掉电方式下，电源电压可以下降到2V，但是为了保存内部 RAM 中的数据，还需要提供约50uA 的电流。

而430单片机功耗是在 uA 级的，工作电流极小，并且超低功耗，关断状态下的电流仅为0.1 μ A，待机电流为0.8 μ A，常规模式下的(250 μ A / 1MIPS@3V)，

端口漏电流不足50 nA，并可零功耗掉电复位(BOR)。另外，该芯片属低电器件，仅需1.8~3.6V 电压供电，因而可有效降低系统功耗。MSP430将低功耗模式扩展为7种，分别对应不同应用场合及任务的低功耗方式。以睡眠模式为例，包括深度睡眠模式 RTC：只有时钟在跑而其他都不动，目前，TI 宣布其MSP430在 RTC 模式下最低功耗仅为360nA。同时也包括诸如液晶显示驱动等需要几十毫秒刷新一次的间歇性睡眠模式。所以，MSP430 的超低功耗更适合应用于使用电池供电的仪器、仪表类产品中。

第三点，51内核单片机由于其内部总线是8 位的，受其结构本身的限制很大，模拟功能控制功能受限制。MSP430 系列其基本架构是16 位的，同时在其内部的数据总线经过转换还存在8 位的总线，在加上本身就是混合型的结构，因而对它这样的开放型的架构来说，无论扩展8 位的功能模块，还是16 位的功能模块，即使扩展模/ 数转换或数/ 模转换这类的功能模块也是很方便的。

第四点，典型 MCU 的存储结构有两种，冯。诺依曼结构——程序存储器和数据存储器统一编码；哈佛结构——程序存储器和数据存储器；MSP430系列单片机属于前者，而常用的51内核单片机系列属于后者。

第五点，就是在开发工具上面。对51内核 单片机来说，不少适合我们使用的开发工具。但是如何实现在线编程还是一个很大的问题。对于 MSP430 系列而言，由于引进了 Flash 型程序存储器和 JTAG 技术，不仅使开发工具变得简便，而且价格也相对低廉，并且还可以实现在线编程。

那么说了以上这些，作为新手，到底怎么学习430这种16位低功耗的单片机

呢？或者说有什么有什么效果比较好的方法吗，答案是肯定。

网上有很多介绍 MSP430单片机学习的方法，一定要结合自己的学习特点，集百家之长，善于总结别人的、自己的学习方法和效果，积极实践。当然学习的方法都是相通的。大概对这些方法简单总结，也就是下面几点：

- 1 作为430新手，首先看我们是否准备好以下几样硬件：电脑(带有并口)1台，MSP430FET 仿真器1套，MSP430开发实验板1套，和一些 MSP430教程或电子教程资料等。
- 2 选用 MSP430仿真器。购买现成的 MSP430JTAG 仿真器如 TI-MSP430JTAG
- 3 选用 MSP430开发板或目标板是学习一个重要的工具，建议购买一个开发板或目标板，同时也会得到很好的技术支持。如果自己制作的话也行，也未尚不可，那就看个人水平了。
- 4 软件工具，因为现在都有学习板的工具软件，如 IAR 的 EW430学习版，430GCC 软件还是免费的。建议还是使用 IAR 软件较方便，因为使用的人群较多，有问题还可请教，容易解决。
- 5 MSP430学习资料及教程，<<MSP430系列16位超低功耗单片机原理与应用>> 清华大学出版社和一本关于430C 语言编程应用的书本<<MSP430系列单片机实用 C 语言程序设计>>。
- 6 MSP430系列中硬件源资是非常丰富的。有看门狗(WDT)、定时器 A (Timer_A)、定时器 B (Timer_B)、比较器、串口0(USART0)、串口1(USART1)、硬件乘法器、液晶驱动器、10位/12位 ADC, 14位 ADC, 数十个可实现方向设置及中断功能的并行输入输出端口、基本定时器 (Basic Timer)。

7 以下几部分模块硬件资源是作为初学者必须要了解学习的。

①WDT 看门狗定时器：新手们会看到很多编程实例中，开头都有“WDTCTL = WDTPW + WDTHOLD;”语句。这是 CUP 执行关闭定时器的意思。在刚开始，我们没必要关心这个，可以跳过这章节，只要大概地知道他的作用就好了。看门狗定时器一般是用于防止程序失效而存在的，一般是用于完整的程序中使用。主程序中可不断地清除看门狗定时器的计数值，以防定时器的值自动计满后使 CPU 复位而重新开始工作。当程序发生错误时，无法正常清除看门狗的定时值时，则看门狗定时器计数溢出而产生 CPU 复位。

②IO 模块：对于 IO 模块可分为初级与提高两阶段，可以不必同一时段内进行理解。

a 初学者必需了解几个常用寄存器使用如 P1OUT, P1IN, P1DIR, P1SEL 寄存器。

b 理解 MSP430 IO 口常用编程语法，这些都可以查看参考例程。

c 结合书本介绍和个人理解，然后在开发板上进行实验。例如比如 LED、响声之类的 初级实验。

d 对于 IO 模块：了解如何使用 IO 中断，和 IO 中断相关寄存器的使用。如果不太必要，可以跳过 IO 中断的使用，而是去学习其他章节的基础部分；当其他章节基础学习到一定程度时回过头再来学习这部分。另外，我们需要搞清楚 IO 模块在什么时候为高阻状态？高阻状态时的漏电流为多少？IO 驱动电流能力如何？等等一些常关注的参数时，我们都需要养成主动查看器件手册的习惯。

1 中断是430处理器的一大特色，因为几乎每个外围模块都能产生，430可以在没有任务时进入低功耗状态，有事件时中断唤醒cpu，处理完毕再次进入低功耗状态。整个中断的响应过程是这样的，当有中断请求时，如果cpu处于活动状态，先完成当前命令；如果处于低功耗，先退出，将下一条指令的pc值压入堆栈；如果有多个中断请求，先响应优先级高的；执行完后，等待中断请求标志位复位，要注意，单中断源的中断请求标志位自动复位，而多中断的标志位需要软件复位；然后系统总中断允许位SR.GIE复位，相应的中断向量值装入pc，程序从这个地址继续执行。这里要注意，中断允许位SR.GIE和中断嵌套问题。如果当你执行中断程序过程中，希望可以响应更高级别的中断请求时，必须在进入第一个中断时把SR.GIE置位。其实，其他的外围模块时钟沿着时钟和中断这个核心来执行的。具体的结构我也不罗嗦了，可以参考430系列手册。

③时钟模块：系统时钟是一个程序运行的指挥官，时序和中断也是整个程序的核心和中轴线。430最多有三个振荡器，DCO内部振荡器；LFXT1外接低频振荡器，常见的32768HZ，不用外接负载电容；也可接高频450KHZ—8M，需接负载电容；XT2接高频450KHZ—8M，加外接电容。初学者若要使用到片中某模块时几乎都要使用不同的时钟。时钟模块是提供整个单片机中各模块的时钟发生源，所以这章节是非常强调地要去认真理解的。

(1). 必须理解430有几种时钟信号：MCLK系统主时钟，可分频1 2 4 8，供cpu使用，其他外围模块在有选择情况下也可使用；SMCLK系统子时钟，供外围模块使用，可选则不同振荡器产生的时钟信号；ACLK辅助时钟，只能由LFXT1产生，供外围模块。

(2). DCO, SMCLK, MCLK, ACLK 各个时钟有什么优点和点。

(3). 4个时钟信号中, 每个时钟的通常频率范围是多少。

(4). 常规的时基控制寄存器设置和时钟如何从引脚输出等等。

④TimaA 模块: (1). 初学者需要搞清 TimerA 的三个不同工作模式中 TA、CCR1、CCR2与 CCR0之间的关系。(2). 搞清楚 TA、CCR1、CCR2与 CCR0之间的中断向量关系。(3). 综合上面的理解, 我们可以结合 TimerA 的例程来进行相关验证, 只有通过亲自的操作才能有效地记住。(4). 利用 TimerA 实现 PWM 信号输出、利用捕获/比较功能实现捕获信号等等。

1 另外多大学习430学习单片机论坛上, 和网友交流, 咨询, 多看别人出现的问题的解决方法。

可以说, 只要经过上述方面认真折腾学习的话, 其实 MSP430单片机编程应用就有了基本入门阶段, 可以说已经闯入 MSP430单片机有效学习过程, 接下要继续学习其他相关的模块应用也不难了。那么剩下的一些模块可以按需而用, 按需而学。但前提的就是需要熟悉几个常用模块基础应用, 以使用量最多的 14x 系列为例, 初学者入门必学有 IO 模块、时基模块、定时器模块等, 以后可继续强化学习如 ADC12模数转换、UART 串行异步通讯模块、比较器 A 模块等等。

(一) 通用 I/O 口的设置-1

1.1 I/O 的简介

特点：①多种复用和设置（即可控制是否输入、是否输出、是否接上拉电阻、是否接下拉电阻、是否可接受中断）；

②一般情况下，P1和P2都是具有中断能力的。从P1和P2接口的各个I/O管脚引入的中断可以独立的被使能并且设置为上升沿或者下降沿触发。对应的中断向量表分别为P1IV和P2IV，它们只能进行字操作，并且PAIV这个寄存器根本不存在。

③P1、P2可合为PA，P3、P4可合为PB，…PC、PD。所以P1为8位BCD 0x00，PA为16位BCD 0x0000。当进行字操作写入PA口时，所有的16位都被写入这个端口；当利用字节操作写入PA口的低字节时，高字节保持不变；

④由于430很多I/O和外围电路接线，所以这里常用位操作。如事先定义（接下来也会用到，先在此声明）BIT0=0x01、BIT1=0x02、BIT3=0x04…BIT7=0x80，那么将P1.1、P1.3的输出设为1的时候，就可以这样操作： $P1OUT |= (BIT1+BIT3)$ 。这样显得很清楚。

⑤没有用到的I/O，要统一拉低为好。此外，当读入的数据长短小于端口最大长度时，那些没有用到的位会被视零。

1.2 I/O 的简单配置

430I/O的配置是用软件来实现的，是通过相应的配置寄存器来实现的。（用到某个I/O时，一定要先配置好该I/O，否则易出错）

1.2.1 I/O 方向设定寄存器 PDIR

如设定P1.1和P1.2为输出状态

操作为： $P1DIR |= (BIT1+BIT2)$ 等价于 $PADIR |= (BIT1+BIT2)$
也等价于 $PADIR_L |= (BIT1+BIT2)$ 。

拉高设定为输出，拉低设定为输入（默认）。

1.2.2 I/O 输入设定寄存器 PXIN

如设定P1.1和P1.2的输入为低电平

操作为： $P1IN \&= \sim (BIT1+BIT2)$ 。

1.2.3 I/O 输出设定寄存器 PXOUT

①当只用为简单的输出时：

如设定P1.1和P1.2输出高电平

操作为： $P1OUT |= (BIT1+BIT2)$ 。

②如果该引脚为正常I/O功能，且当前已设定为输入方向，且上拉/下拉电阻寄存器是有效地。那么PXOUT可以用来配置上拉和下拉电阻：

低电平为下拉电阻；

高电平为上拉电阻；

1.2.4 上拉/下拉电阻使能寄存器 PXREN

低电平该寄存器为无效状态；

高电平该寄存器为有效状态；

1.2.5 输出驱动能力设置寄存器 **PXDS**

弱化驱动可以减弱电磁干扰 EMI，全力驱动会增强电磁干扰。默认为减弱驱动。

低电平表示减弱的驱动（默认）；

高电平表示全力的驱动；

1.2.6 功能选择寄存器 **PXSEL**

用来声明该端口是要应用于外围电路的特殊功能（不决定输入输出方向），默认为低电平。

低电平表示普通的 I/O（默认）；

高电平表示该引脚将有连接外围电路的特殊用途；

如：开发板初始化函数 HAL_Board.c 中有这样一句程序：

```
P5SEL |= (BIT2+BIT3) (=00001100);
```

这句话的意思就是声明 P5.2和 P5.3将有特殊用途，实际上这两个 I/O 接的是外部的高频时钟晶振（之后还要设定为输入状态才可以）。

此外需要注意的是，一旦某个 I/O 的 PXSEL 置高了，那么该引脚将不能再被用为中断引脚。

总结，简单的程序应用：

/*实现 LED 的闪烁*/LED 位于每个触摸按键下方，具体接口请查询原理图

#include <msp430.h> 该头文件内部包含430各个寄存器的配置情况

void main(void)

{

 WDTCTL=WDTPW+WDTHOLD; **//关闭看门狗**

 P1DIR|=(BIT0+BIT1+BIT2+BIT3+BIT4+BIT5); **//P1.0-P1.5方向为输出，**

BITX 的定义在 msp430.h 中

 P1OUT&=~(BIT0+BIT1+BIT2+BIT3+BIT4+BIT5); **//清零**

/*P1SEL=0X00;

PXDS=0X00;默认*/

 int i=0,j=0;

 while(1)

 {

 if(i>5)

 i=0;

 else

 {

 switch(i)

 {

 case 0:P1OUT=0x01;break;


```

        case 1:P1OUT=0x02;break;
        case 2:P1OUT=0x04;break;
        case 3:P1OUT=0x08;break;
        case 4:P1OUT=0x10;break;
        case 5:P1OUT=0x20;break;
    }
}
i++;
for(j=20000;j>0;j--);           //延时
}
}

```

(一)通用 I/O 口的设置-2

这一部分讲外部中断。

看介绍再加看懂程序才是王道

外部中断是 **MSP430** 优先级最低的中断而且是可屏蔽中断。用起来比较简单。

1.2.7 简单的端口中断（外部中断）

P1、P2的所有端口都具有中断能力，可以通过寄存器 **PxIFG**,**PxIE** 和 **PxIES** 来配置。其他端口则需参照具体的引脚说明书。所有的 **P1**中断标志是最优先的（相比其它引脚的外部中断），其中 **P1IFG.0**又最优。

PXIV 中断向量寄存器：只有 **P1IV** 和 **P2IV**。最高优先级使能中断在 **P1IV** 寄存器中产生一个序号，这个数字会被程序计数器识别或者加入其中，然后自动的执行合适的中断服务程序。关闭 **P1**口中断不会影响 **P1IV** 寄存器中的值。**P2**口具有相同的功能。**PxIV** 寄存器只能字访问。

PxIFGx 中断标志寄存器：只有相应的中断使能 **PXIE** 打开且总中断 **GIE** 打开，该寄存器才有效。

低电平表示没有中断请求等待响应；

高电平表示有中断请求等待响应；

注意：使用端口的中断功能期间，如果进行 **PXIN**、**PXOUT** 等操作可能使中断变化。

注意：中断标志需要软件清零。有一种情况例外：两个中断同时发生，先响应优先级高的中断，当该中断服务程序结束后，该位的中断标志会自动清零，然后去响应另外一个中断。

PxIE 中断使能寄存器

低电平表示中断关闭；

高电平表示中断允许；

PXIES 中断触发方式选择寄存器

低电平表示上升沿触发；

高电平表示下降沿触发；

外部中断应用示例：/*采用中断的方式，开关 S2（接于 P2.2）控制 LED（接于 P1）一个一个点亮（看 PCB 图接线）*/

```
#include <msp430.h>
```

```
int s=0; //s 用来表示按键次数
```

```
int num =0; //num表示 LED 值
```

```
void main(void){
```

```
    WDTCTL=WDTPW+WDTHOLD; //关闭看门狗
```

```
    P1DIR=0xff; //P1全部接输出
```

```
    P1OUT=0x00; //接 LED 初始化故全拉低，这样开始灯是灭的
```

```
    P2DIR=0x00; //P2全部设为输入。因为要接受外部中断
```

```
    P2IFG=0x00; //清除 P2口的中断标志
```

```
    P2IE=BIT2; //P2.2开启中断
```

```
    P2IES=0xff; //P2为下降沿触发
```

```
    P2IN=BIT2; //P2.2输入拉高，因此开关闭合时会拉低产生下降沿（即中断）
```

```
    P2OUT=0xff;
```

```
    P2REN=0xff; //作为输入的时候一定要配置上拉电阻（很重要，容易忘记，我就在这错了-_-）
```

```
    __enable_interrupt(); //开总中断
```

```
    while(1)
```

```
{
```

```
    num=s%5;
```

```
    switch(num){
```

```
        case 0:P1OUT=BIT1;break;
```

```
        case 1:P1OUT=BIT2;break;
```

```
        case 2:P1OUT=BIT3;break;
```

```
        case 3:P1OUT=BIT4;break;
```

```
        case 4:P1OUT=BIT5;break;}}}
```

```
#pragma vector=PORT2_VECTOR //固定格式，声明中断向量地址
```

```
__interrupt void Port2_ISR(void) { //中断子程序
```

```
    unsigned int temp; //局部变量
```

```
    int i;
```

```
    for(i=0;i<12000;i++); // 延时消抖
```

```
    if((P2IN&0xff)!=0xff){ //如果有键按下
```

```
        temp=P2IFG; //读取中断标志
```

```
        P2IFG=0x00; //标志位清零
```

```
        if(temp==0x04) //如果 P2.2产生中断
```



```

        s++;}} //这一部分其实有几句话很多余，主要是为了体现
每个端口中断的知识
备注中断子程序调用格式：
#pragma vector=中断向量地址
__interrupt void 中断服务程序名(void)
{
//中断处理程序
}

```

1.2.8 未使用的 I/O

未使用的 I/O 管脚最好被设置为普通 I/O 功能、输出方向并且在 PCB 板上不连接这些管脚，以防止浮动的输入和降低功耗。因为这些管脚没有被连接，所以它们的输出值没有必要在意。或者可以通过设置未使用管脚的 PxREN 寄存器来使能置高/置低寄存器以避免浮动输入的干扰。

(二)看门狗的设置

看门狗定时器 (Watchdog Timer(WDT_A)) 实际上是一个特殊的定时器，即可以用来作为看门狗使用，也可以用作定时器。

所谓的看门狗功能，是指可以监控程序是否由于某些干扰或者错误而跑飞。其原理就是发生故障的时间满足规定的定时时间后，产生一个非屏蔽中断，使系统复位。这样当在调试程序或预计程序在某个地方可能瞬时发生错误时（如外部电路干扰），选用设置看门狗定时中断可以避免程序跑飞。

当然，它也可以用作一般的定时功能。

不过实际上，由于看门狗定时器（作看门狗使用时）需要很严密的设置（否则程序容易经常重启），所以很多人都不会使用这项功能。所以，程序一开始就加上一句话：WDTCTL=WDTPW+WDTHOLD 来关闭看门狗。

2.1 WDT_A 的简介

特点：

- ①有8种可选定时时间；
- ②看门狗模式；
- ③定时器模式；
- ④看门狗控制寄存器存在密码保护；
- ⑤时钟源可选，且具有时钟源意外保护；
- ⑥可以被终止来节省能源；
- ⑦无论是用作看门狗还是定时器，其间隔时间都无法随意设定，只能从8种设定中选择，当然可以通过更改时钟频率来间接更改时间；

注意默认设置：程序启动，看门狗即启动；监控周期为

32ms/32.768KHZ(也就是说当看门狗的时钟频率为32.768KHZ 时，每过32ms，如果不软件清空，程序就会重启)；所用的时钟源为 SMCLK（实际频率不是32.768KHZ，后面会提到）。

2.2 WDT_A 的寄存器及操作

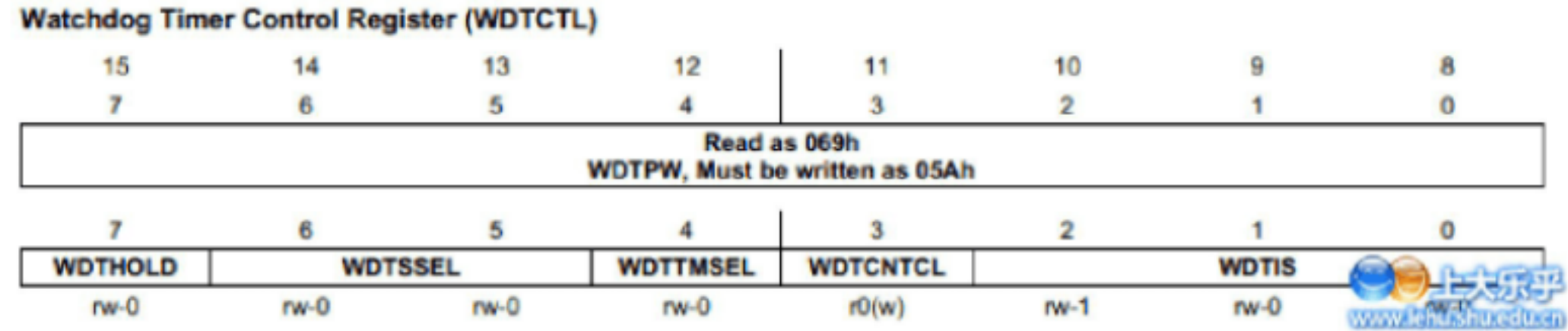
注意：所有的寄存器都存字操作和字节操作模式。例如对 WDTCTL 寄存器直接赋值是字操作，你也可以通过 WDTCTL_L（低字节寄存器）、WDTCTL_H 来进行字节操作赋值。

该类寄存器只能赋值不要进行“|=、&=”等逻辑操作。

2.2.1看门狗控制寄存器 WDTCTL（Watchdog Timer Control）

该寄存器是一个16位带有密码保护的读写寄存器。所谓的密码保护，是为了防止该寄存器被不小心篡改。那么是如何实现密码保护呢？事实上，该寄存器的高字节用来存放口令，低字节才是控制数据。写口令为05Ah，读口令为069h，任何和口令不一样的高字节操作都会导致系统复位。

该寄存器的位功能定义如下：（支持位操作）




WDTPW: Bits15-8, WDT Password, 写为05Ah，读为069h。

WDTHOLD: Bit7, WDT HOLD,

- 0: 打开看门狗计时器;
- 1: 关闭

WDTSEL: Bits6-5, WDT Clock Source Select 时钟源选择

- 00 SMCLK
 - 01 ACLK
 - 10 VLOCLK
 - 11 X_CLK; VLOCLK in devices that do not support XCLK
- 

WDTTMSEL: Bit4, 工作模式选择

- 0: 看门狗模式;
- 1: 定时器模式;

WDTCNTCL: Bit3, 计时器清零

- 0: 无效;
- 1: 清空计数器, 即 WDCNT=0x0000h

WDTIS: Bits2-0, WDT Interval Select, 间隔时间选择。用来选择计数的周期，共有8种时间可以选择。

备注：括号前数字为计数器的值

000	Watchdog clock source /2G (18:12:16 at 32 kHz)
001	Watchdog clock source /128M (01:08:16 at 32 kHz)
010	Watchdog clock source /8192k (00:04:16 at 32 kHz)
011	Watchdog clock source /512k (00:00:16 at 32 kHz)
100	Watchdog clock source /32k (1 s at 32 kHz)
101	Watchdog clock source /8192 (250 ms at 32 kHz)
110	Watchdog clock source /512 (15,6 ms at 32 kHz)
111	Watchdog clock source /64 (1.95 ms at 32 kHz)



2.2.2 看门狗计数值寄存器 **WDT CNT** (Watchdog Timer Counter)

这是一个32Bit 的加计数器，但是不能通过软件来直接对其进行赋值等操作，只能通过 WDTCTL 中的 WDTIS 来选择时间。或者你可以通过 WDTSEL 选择不同的时钟源，来间接改变时间。

2.2.3 看门狗中断 位控制

WDT 利用 SFRS 总寄存器中的两位来控制中断。

WDT 中断标志位: **WDTIFG**，位于 SFRIFG1.0

WDT 中断使能位: **WDTIE**，位于 SFRIE1.0

看门狗模式: 如果不及时对 WDT CNT 清零或者对 WDT 初始化，WDTIFG 就会置位，然后程序就会重启。

计时器模式: 必须开总中断 GIE 和看门狗中断 WDTIE。此外，当执行完中断服务程序后，标志位 WDTIFG 会自动清零。

2.3 常用操作

*/*关闭看门狗*/*

WDTCTL=WDTPW+WDT HOLD;

*/*喂狗，即对有效状态的看门狗进行清零操作*/*

/ 时间计满还不喂狗就会导致程序重启*/*

*/*WDTIS2即 WDTIS=100，此时时间间隔设为1S，假定频率为32.768KHZ*/*

WDTCTL = WDTPW + WDT CNTCL+WDTSEL0+WDTIS2;

*/*将看门狗设置为计数器模式，计数8192约250ms，假定频率为32.768KHZ*/*

/ WDTIS2+WDTIS0即101*/*

WDTCTL=WDTPW+WDT CNTCL+WDTTMSSEL+WDTIS2+WDTIS0

总结例程:

*/*首先将 WDT 设为定时器功能。而中断服务子程序则是把 WDT 改为看门狗功能。这样就通过中断控制的周期性的重启程序，实现了 LED 闪烁*/*

/*注意，这里默认的时钟源不是32KHZ，而是内部的 DCO-SMCLK（之后会讲到）
1.045MHZ。所以前面 WDTIS 定义的时间大约都要缩小1045/32=32倍*/

```
#include <msp430.h>
void main(void)
{
    /*清零-设定为计数器-时间设定为010模式，即256S/32=8S*/
    WDTCTL=WDTPW+WDTCNTCL+WDTTMSSEL+WDTIS1;
    __enable_interrupt(); //开总中断
    SFRIE1|=WDTIE;        //开看门狗定时器中断
    P1DIR=0xff;
    int i,j;
    P1OUT=0xff;
    for(i=0;i<30000;i++)
        for(j=0;j<50;j++);    //延时约8S
    P1OUT=0x00;
    while(1);
}

/*中断服务程序*/
#pragma vector=WDT_VECTOR
__interrupt void WatchTimer(void)
{
    WDTCTL=WDTPW+WDTCNTCL+WDTIS1;    //看门狗模式，时间设定约为8S
}
```

MSP430F5529 番外(一)常用内置函数和一些说明

(1) MSP430F5529支持最高工作频率为25MHZ，也就是说你通过锁相环倍频来提高系统运行速度是有一个限制的，最高只能到25MHZ（再高没意思了）。

(2) 几个重要的内联函数

（内联函数定义在 intrinsics.h 中，但是这几个函数使用的时候不需要事先声明该文件头）

① __bic_SR_register();

将 CPU 中 SR 寄存器中的某些位清零。即将括号内的位清零。

例：__bic_SR_register(GIE);//将 GIE 位清零，即关闭总中断

② __bis_SR_register();

将 CPU 中 SR 寄存器中的某些位置1。即将括号内的位置1。

③ __interrupt

放在函数前面，标志中断函数。下面这段程序是看门狗

WDT 的中断服务函数。WDT_VECTOR 为看门狗的中断向量。 举例：

```
#pragma vector=WDT_VECTOR
__interrupt void WatchDog(void)
{... }
```

④ **__monitor**

放在函数前面，功能是声明当这一函数执行的时候自动关闭中断。应该尽量缩短这样的函数，否则，中断事件无法得到及时的响应。

⑤ **__bic_SR_register_on_exit();**

功能：用于一个中断函数或者不可中断函数（标志为__monitor）返回时，将 CPU 内 SR 寄存器中的某些位清0。

⑥ **__bis_SR_register_on_exit();**

功能：用于一个中断函数或者不可中断函数（标志为__monitor）返回时，将 CPU 内 SR 寄存器中的某些位置1。

⑦ **__no_init**

放在全局变量前面，功能是使程序启动时不为变量赋初值

⑧ **__disable_interrupt**

关闭总中断

另外一个相同作用的表达为： **_DINT()**

⑨ **__enable_interrupt**

开启总中断

另外一个相同作用的表达为： **_EINT()**

⑩ **__even_in_range(,);**

常被用在多源中断的查询中，如 **switch(__even_in_range(TAIV, 10)**
意思是：只有在 TAIV 的值是在0-10内的偶数时才会执行
switch 函数内的语句。其作用是提高 **switch** 语句的效率

A. _NOP()

空操作，等价于__no_operation 指令

B. __get_SP_register(void)

功能：返回堆栈指针寄存器 SP 的值。

C. __get_SR_register_on_exit(void)

功能：用于一个中断函数或者不可中断函数
（标志为__monitor）返回时，返回状态寄存器 SR 的值。

D. __bcd_add_short(unsigned short,unsigned short);

功能：两个16为 BCD 格式的数字相加，返回和。

E. __bcd_add_long(unsigned long,unsigned long);

功能：两个32位 BCD 格式的数字相加，返回和。

F.__delay_cycles(x);

系统自带的精确延时。x 必须是常量或则是常量表达式，如果是变量则编译报错！延时的时间为 x 乘以 MCLK 的时钟周期

(3) 关于 MSP430 大量寄存器如何处理的问题：

MSP430 寄存器太多了，把每一个都记住实在太难。所以，我的建议是，学习的时候，把重要的常用的寄存器记住。

至于其它众多寄存器，大家只需要有一个印象，知道这些寄存器可以控制那些设置。到具体用的时候，再查找技术手册；

(4) 头文件 msp430f5529.h

里面不仅定义了对各个寄存器的声明，还定义了很多很方便的东东。

比如你要进入低功耗模式1：LPM1；

比如你要把定时器 A0 的时钟选为 SMCLK；

最原始的做法是 TA0CTL=0x0200，但现在有更清晰的做法
TA0CTL=TASSEL_2; //时钟源选择模式2

所以说要不断发掘不断积累！

(5) 中断寄存器的名字

大家都知道中断函数是怎么写的，模式为：

```
#pragma vector=中断向量地址（名）
__interrupt void 自定义中断函数名(void)
{...}
```

但是想要写出来，你首先得知道中断向量叫什么名字吧

到目前，我们已经遇到了看门狗中断 WDT_VECTOR、引脚 P2 的外部中断 PORT2_VECTOR、定时器 A0（CCR1-CCR4 和 TAIFG）中断 TIMER0_A1_VECTOR 还有大量的中断向量，我们目前还不知道名字该怎么办。

打开 msp430f5529.h，拉到该文件底部，就展示了定义的各种中断向量名，还有解释。

(6)

MSP430 的最大特色就是低功耗，这个是体现在各个方面的。在整体层次上，MSP430 可以设置整个系统的工作模式以达到适应工作要求且降低功耗的目的。

为了降低功耗，处理器有几个考虑：

一个是降低工作电压（F5529 为 3.3V 很低了吧，而且内部核心电压 V_{CORE} 更低）；

第二个就是把暂时不用的模块功能关闭掉（F5529 各个模块都可以独立运行，如定时器、A/D 转换、看门狗等都可以在 CPU 休眠的状态下独立工作。若需要主 CPU 工作，任何一个模块都可以通过中断唤醒 CPU，从而使系统以最低功耗运行。

）；

第三个方法就是降低工作时钟频率。

MSP430F5529 （三）统一时钟系统 UCS-1

之前有一点漏说明了，先补充上

```
*****
****还有一个模块时钟源：MODOSC，产生 MODCLK 时钟源信号，一般
只为闪存控制模块和 ADC12模块提供服务。
该模块不被使用时自动关闭，任何模块对该时钟源提出使用要求时，
MODOSC 无需被使能即可响应该请求。430F5529中 MODCLK 为5MHZ。
*****
***
```

MSP430F5529有多个时钟源，而且很多模块其时钟源都是可以自由选择的。此外，由于一般情况下，系统功耗是和工作频率成正比的，因此有些时候通过选择较低频率的时钟源，在满足正常工作条件下，是可以有效降低功耗的。虽然函数库 [HAL_UCS.c/h](#)，有完整的各个控制函数，但我觉得对于这一章还是对寄存器直接操作比较简单，因为函数太短、太多了。

3.1统一时钟系统（UCS）的简介

Unified Clock System，UCS。合理的配置时钟，可以达到平衡系统且降低功耗的目的。

MSPF5529时钟系统包含5个时钟源：

- ①LFXT1 外部低频振荡源，32.768KHZ，可以用作 FLL 的参照源；
- ②XT2 外部高频振荡源，4MHZ；
- ③VLO （Internal very low）内部低耗低频振荡源，典型为10KHZ，精度一般；
- ④REFO 内部低频参照源，32.768KHZ，常被用作锁相环 FLL 的基准频率，精度很高，不使用时不消耗电源，其设置往往要参考 LPM 模式的设置；
- ⑤DCO （Internal digitally-controlled）内部数字控制振荡源，一般通过 FLL 来设置；（很有用，很重要，之后会详细讲）

通常使用3种时钟信号，它们都来自于上述5个信号源：

- ①ACLK （Auxiliary clock）辅助时钟，其时钟源可由软件控制从 XT1、REFO、VLO、DCO、DCOCLKDIV、XT2里面选取。其中 DCOCLKDIV 是由 DCO 经1、2、4、8、16或者32分频得到。注意，ACLK 同样可以再次被1、2、4、8、16或者32分频。
- ②MCLK （Master clock）主时钟，其特性与 ACLK 一模一样。
- ③SMCLK （Subsystem master clock）子系统时钟，其特性与 ACLK 一模一样。

3.2 UCS 的操作说明

开机上电时默认的时钟情况为（必须记清楚!!!!）：

ACLK: XT1（无效时，低频模式切换为 REFO，其他情况切换为 DCO）

MCLK: DCOCLKDIV

SMCLK: DCOCLKDIV

此外，FLL 的参照源默认 XT1；

如果连接 XT1和 XT2的引脚不进行 PXSEL 的设置，那么这两个时钟源都是无效的；

REFCLK、VLOCLK、DCOCLK 默认状态下是可用的；

系统稳定后，DCOCLK 默认为2.097152MHZ，FLL 默认2分频，则 MCLK 和 SMCLK 的频率都为1.048576MHZ。（实验三会提到如何计算）

另外，系统复位、系统工作模式 LPM 的选择都会对 UCS 有一定影响，这里限制太多，具体可参考 TI 官方资料 UCS 部分。LPM 以及系统复位下章将会讲到。

关于操作说明的简单总结：（下面基本都是废话，了解即可）

①VLO 的选择是最简单的，不需要顾及其它情况；

②REFO 的选用，需要参考不同的工作模式，有多种限制；

③XT1和 XT2特点相同。使用的时候，不仅要配置与其相连的引脚，还要配置电容，还要注意其本身工作在低频还是高频模式。而且，在不同工作模式下也有不同的要求；

④DCO 作为数控振荡器，其频率的调节不仅可以通过自身设定，也可以通过 FLL 锁相环设定；

⑤FLL 锁相环，是变换频率的灵活选择。它既可以设置基准频率，也可以选择分频数，还可以被直接关闭来实现降低功耗等目的；

⑥UCS 系统带有时钟信号错误保护机制；

⑦对有严格时序要求的地方，要选择精度高的时钟源，并且做好 FLL 和 DCO 部分的调制设置；

⑧不同模式下（有些时钟源是禁止的）的时钟控制图：（只需用到的时候注意一下即可，查表）

Table 5-1. Clock Request System and Power Modes

Mode	ACLK		MCLK		SMCLK			
	ACLKREQEN = 0	ACLKREQEN = 1	MCLKREQEN = 0	MCLKREQEN = 1	SMCLKOFF = 0		SMCLKOFF = 1	
					SMCLKREQEN = 0	SMCLKREQEN = 1	SMCLKREQEN = 0	SMCLKREQEN = 1
AM	Active	Active	Active	Active	Active	Active	Disabled	Active
LPM0	Active	Active	Disabled	Active	Active	Active	Disabled	Active
LPM1	Active	Active	Disabled	Active	Active	Active	Disabled	Active
LPM2	Active	Active	Disabled	Active	Disabled	Active	Disabled	Active



	ACLK		MCLK		SMCLK			
	Active	Active	Disabled	Active	Disabled	Active	Disabled	Active
LPM3	Disabled	Active	Disabled	Active	Disabled	Active	Disabled	Active
LPM4	Disabled	Active	Disabled	Active	Disabled	Active	Disabled	Active
LPM3.5 (1)	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled
LPM4.5 (1)	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled	Disabled

3.3 UCS 寄存器控制操作

共有10组16位读写寄存器，为 UCSCTL0-UCSCTL9。同样支持字和字节操作，即 UCSCTL0包括 UCSCTL0_H 和 UCSCTL0_L。

注：凡是标记“Reserved”的位，如果没有特意声明，则读回时都按0处理。

UCSCTL0

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved				DCO			
r0	r0	r0	rw-0	rw-0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
MOD					Reserved		
rw-0	rw-0	rw-0	rw-0	rw-0	r0	r0	

DCO：DCO 频拍选择。选择 DCO 的频拍并在 FLL 运行期间（因 MOD 位的变化）自动调整。。DCO 的5个控制位把由 DCORSELx 选择的 DCO 频率分为32等份，间隔大约8%。

MOD：调制位计数器。选择调制类型，所有的 MOD 位在 FLL 运行期间自动调整，无需用户干预。

UCSCTL1

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved	DCORSEL			Reserved	Reserved	Reserved	DISMOD
r0	rw-0	rw-1	rw-0	r0	r0	rw-0	rw-0

DCORSEL：DCO 频率范围选择

DISMOD：调制器禁止使能位。0—使能调制器；1—禁止调制器。

UCSCTL2

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved	FLLD			Reserved		FLLN	
r0	rw-0	rw-0	rw-1	r0	r0	rw-0	rw-0
7	6	5	4	3	2	1	0
FLLN							
rw-0	rw-0	rw-0	rw-1	rw-1	rw-1	rw-1	

FLLD: 预分频器（即 f_{DCO} 分频）。000-1分频，001-2分频，010-4分频，011-8分频，100-16分频，101-32分频，110以及111都是备用的，默认为32分频。

FLLN: 倍频系数。设置倍频值 N，N 必须大于0，如果 FLLN=0，则 N 被自动设置为1。

UCSCTL3

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved	SELREF			Reserved	FLLREFDIV		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	

SELREF: FLL 参考时钟选择。

000-XT1，001-待用，默认为 XT1，010-REFO，101-XT2，其余均为待用，默认为 REFO。

FLLREFDIV: FLL 参考时钟分频器。000-1分频，001-2分频，010-4分频，011-8分频，100-12分频，101-16分频，110以及111都是备用的，默认为16分频。

UCSCTL4

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved				SELA			
r0	r0	r0	r0	r0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
Reserved	SELS			Reserved	SELM		
r0	rw-1	rw-0	rw-0	r0	rw-1	rw-0	

SELA: ACLK 时钟源选择。

000-XT1，001-VLO，010-REFO，011-DCO，100-DCOCLKDIV，101-XT2有效时为 XT2，否则为 DCOCLKDIV
110、111保留以备后来使用。当 XT2有效时默认为 XT2CLK，否则默认为 DCOCLKDIV

SELS: SMCLK 时钟源选择。设置同 SELA

SELM: MCLK 时钟源选择。设置同 SELA

UCSCTL5

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved	DIVPA			Reserved	DIVA		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	rw-0
7	6	5	4	3	2	1	0
Reserved	DIVS			Reserved	DIVM		
r0	rw-0	rw-0	rw-0	r0	rw-0	rw-0	

DIVPA: ACLK 外部有效输出分频000-1分频, 001-2分频, 010-4分频,
011-8分频, 100-16分频, 101-32分频,
110以及111都是备用的, 默认为32分频。

DIVA: ACLK 时钟源分频, 设置同 DIVPA

DIVS: SMCLK 时钟源分频, 设置同 DIVPA

DIVM: MCLK 时钟源分频, 设置同 DIVPA

UCSCTL6

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
XT2DRIVE		Reserved	XT2BYPASS	Reserved		XT2OFF	
rw-1	rw-1	r0	rw-0	r0	r0	r0	rw-1
7	6	5	4	3	2	1	0
XT1DRIVE		XTS	XT1BYPASS	XCAP		SMCLKOFF	XT1OFF
rw-1	rw-1	rw-0	rw-0	rw-1	rw-1	rw-0	

XT2DRIVE: XT2振荡器电流驱动能力调整

00 最低电流消耗。XT2振荡器工作在4MHz 到8MHz ...

XT2BYPASS: XT2旁路选择 0-XT2来源于内部时钟（使用外部晶振）

1-XT2来源于外部引脚输入（旁路模式）

XT2OFF: 关闭 XT2振荡器

0 -当 XT2引脚被设置为 XT2功能且没有被设置位旁路模式时, XT2被打开;

1 -当 XT2没有被用作时钟源以及没有用作 FLL 参考时钟时, XT2被关闭。

XTS: XT1工作模式选择

0-低频模式（XCAP 定义 XIN 和 XOUT 引脚间的电容）

1-高频模式（XCAP 位没有被使用）

XCAP: 振荡器负载电容选择

SMCLKOFF: SMCLK 关闭控制位 0-SMCLK 开 1-SMCLK 关闭

XT1OFF: 同 XT2OFF

UCSCTL7

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved				Reserved		Reserved	
r0	r0	rw-0	rw-(0)	rw-(1)	rw-(1)	r-1	r-1
7	6	5	4	3	2	1	0
Reserved				XT2OFFG ⁽¹⁾	XT1HFOFFG ⁽¹⁾	XT1LFOFFG	DCOFFG
r0	r0	r0	rw-(0)	rw-(0)	rw-(0)	rw-(1)	rw-(1)

XT2OFFG: XT2出错时置位，同时 OFFIFG 也会置位，需要软件清零。

XT1HFOFFG: 高频工作模式下 XT1出错时置位，同时 OFFIFG 也会置位，需要软件清零。

XT1LFOFFG: 低频工作模式下 XT1出错时置位，同时 OFFIFG 也会置位，需要软件清零。

DCOFFG: DCO 出错时置位，但当 DCO=1或31时，也会置位，同时 OFFIFG 也会置位，需要软件清零。

UCSCTL8

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved				Reserved			
r0	r0	r0	r0	r0	rw-(1)	rw-(1)	rw-(1)
7	6	5	4	3	2	1	0
Reserved				MODOSC REQEN	SMCLKREQEN	MCLKREQEN	ACLKREQEN
r0	r0	r0	rw-(0)	rw-(0)	rw-(1)	rw-(1)	rw-(1)

信号请求使能：

0-相应的信号请求禁止1-相应的信号请求允许

UCSCTL9

15	14	13	12	11	10	9	8
7	6	5	4	3	2	1	0
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved						XT2BYPASSL V	XT1BYPASSL V
r0	r0	r0	r0	r0	r0	rw-0	rw-0

XT1、XT2旁路模式输入摇摆电平（范围）必须被设置

0-输入范围0~DV_{CC}

1-输入范围0~DV_{IO}

3.4实验总结

实验一：将 MCLK 和 SMCLK 配置为 REFOCLK、VLOCLK(需要示波器测量)

/* REFOCLK 和 VLOCLK 是芯片默认提供的，只要芯片正常工作，这两个时钟就会

正常工作，因此，该时钟配置非常简单，只需要修改 UCSCTL4，将 SELS 和 SELM 配置为对应的选项 VLOCLK 或者 REFOCLK 即可*/

```
#include
void main(void){
    WDTCTL = WDTPW+WDTHOLD;
    P1SEL |= BIT0;//声明有特殊功能，将不被用作普通 I/O
    P1DIR |= BIT0;//ACLK 输出端，用来测量 ACLK 频率，外接频率计测
    P2SEL |= BIT2;P2DIR |= BIT2;//SMCLK 输出端
    P7SEL |= BIT7;P7DIR |= BIT7;//MCLK 用输出端
    //UCSCTL4 = UCSCTL4&(~(SELS_7|SELM_7))|SELS_1|SELM_1; //将 SMCLK
    和 MCLK 配置为 VLOCLK
    UCSCTL4 = UCSCTL4&(~(SELS_7|SELM_7))|SELS_2|SELM_2; //将 SMCLK 和
    MCLK 配置为 REFOCLK
    /* UCSCTL4&(~(SELS_7|SELM_7))这一语句相当于先把 SELS 和 SELM 清零*/
    while(1);
}
```

MSP430F5529 （三）统一时钟系统 UCS-2

实验二：将 MCLK 和 SMCLK 配置为 XT1（F5529的 XT1为32.768KHZ）

```
/*1.配置 IO 口5.4和5.5为 XT1功能。*/
/*2.配置 XCAP 为 XCAP_3，即12PF 的电容。*/
/*3.清除 XT1OFF 标志位。*/
/*4.等待 XT1起振。*/
#include <msp430.h>
void main(void){
    P1SEL |= BIT0;P1DIR |= BIT0;//测量 ACLK 用
    P2SEL |= BIT2;P2DIR |= BIT2;//测量 SMCLK 用
    P7SEL |= BIT7;P7DIR |= BIT7;//测量 MCLK 用
    P5SEL |= BIT4|BIT5; //配置为 XT1功能,电路板上晶振接于这两脚
    UCSCTL6 |= XCAP_3; //配置电容为12pF
    UCSCTL6 &= ~XT1OFF; //使能 XT1
    /*下面是很重要的一步：*/
    /* XT1刚刚起振的时候可能有错误,导致时钟错误标志位置位,必须先清零*/
    /*OFIFG 即 Osc Fault Flag, 位于 SFRIFG1中*/
    while(SFRIFG1 & OFIFG) //如果有时钟错误{
        UCSCTL7 &= ~(XT2OFFFG+DCOFFFG+XT1LFOFFFG); //清除3种时钟错误标志
        SFRIFG1&= ~(OFIFG); //清除时钟错误标志位}
    UCSCTL4&=(UCSCTL4&(~(SELS_7|SELM_7)))|SELS_0|SELM_0;
```

//将 SMCLK 和 MCLK 时钟源配置为 XT1}

实验三：DCO-FLL 数控振荡器结合锁相环

DCO 模块在 MSP430F5529 系列芯片中非常重要，因为从 MSP430F4XX 开始，MSP430 引用了 FLL 模块，FLL 即锁相环，可以通过倍频的方式提高系统时钟频率，进而提高系统的运行速度。

DCO 模块运行需要参考时钟 REFCLK，REFCLK 可以来自 REFOCLK、XT1CLK 和 XT2CLK，通过 UCSCTL3 的 SELREF 选择，默认使用的 XT1CLK，但如果 XT1CLK 不可用则使用 REFOCLK。

DCO 模块有两个输出时钟信号，即 DCOCLK 和 DCOCLKDIV，其中，倍频计算公式如下：

$$\text{DCOCLK} = D * (N + 1) * (\text{REFCLK} / n)$$

$$\text{DCOCLKDIV} = (N + 1) * (\text{REFCLK} / n)$$

其中：n 即 REFCLK 输入时钟分频，可以通过 UCSCTL3 中的 FLLCLKDIV 设定，默认为 1，也就是不分频；

D 可以通过 UCSCTL2 中的 FLLD 来设定，默认为 1，也就是 2 分频；

N 可以通过 UCSCTL2 中的 FLLN 来设定，默认值为 32。

所以，系统上电后如果不做任何设置，DCOCLK 的实际值为 2097152，DCOCLKDIV 的实际值为 1048576。

另外，配置芯片工作频率还需要配置 DCORSEL 和 DCOx，DCORSEL 和 DCOx 的具体作用如下：

DCORSEL 位于 UCSCTL1，共 3 位，将 DCO 分为 8 个频率段。

DCOx 位于 UCSCTL0 共 5 位，将 DCORSEL 选择的频率段分为 32 个频率阶，每阶比前一阶高出约 8%，该寄存器系统可以自动调整，通常配置为 0。

下表给出了相应设置情况下的频率调节范围：

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
$f_{\text{DCO}(0,0)}$	DCO frequency (0, 0)	DCORSELx = 0, DCOx = 0, MODx = 0	0.07		0.20	MHz
$f_{\text{DCO}(0,31)}$	DCO frequency (0, 31)	DCORSELx = 0, DCOx = 31, MODx = 0	0.70		1.70	MHz
$f_{\text{DCO}(1,0)}$	DCO frequency (1, 0)	DCORSELx = 1, DCOx = 0, MODx = 0	0.15		0.36	MHz
$f_{\text{DCO}(1,31)}$	DCO frequency (1, 31)	DCORSELx = 1, DCOx = 31, MODx = 0	1.47		3.45	MHz
$f_{\text{DCO}(2,0)}$	DCO frequency (2, 0)	DCORSELx = 2, DCOx = 0, MODx = 0	0.32		0.75	MHz
$f_{\text{DCO}(2,31)}$	DCO frequency (2, 31)	DCORSELx = 2, DCOx = 31, MODx = 0	3.17		7.38	MHz
$f_{\text{DCO}(3,0)}$	DCO frequency (3, 0)	DCORSELx = 3, DCOx = 0, MODx = 0	0.64		1.51	MHz
$f_{\text{DCO}(3,31)}$	DCO frequency (3, 31)	DCORSELx = 3, DCOx = 31, MODx = 0	6.07		14.0	MHz
$f_{\text{DCO}(4,0)}$	DCO frequency (4, 0)	DCORSELx = 4, DCOx = 0, MODx = 0	1.3		3.2	MHz
$f_{\text{DCO}(4,31)}$	DCO frequency (4, 31)	DCORSELx = 4, DCOx = 31, MODx = 0	12.3		28.2	MHz
$f_{\text{DCO}(5,0)}$	DCO frequency (5, 0)	DCORSELx = 5, DCOx = 0, MODx = 0	2.5		6.0	MHz
$f_{\text{DCO}(5,31)}$	DCO frequency (5, 31)	DCORSELx = 5, DCOx = 31, MODx = 0	23.7		54.1	MHz
$f_{\text{DCO}(6,0)}$	DCO frequency (6, 0)	DCORSELx = 6, DCOx = 0, MODx = 0	4.6		10.7	MHz
$f_{\text{DCO}(6,31)}$	DCO frequency (6, 31)	DCORSELx = 6, DCOx = 31, MODx = 0	39.0		88.0	MHz
$f_{\text{DCO}(7,0)}$	DCO frequency (7, 0)	DCORSELx = 7, DCOx = 0, MODx = 0	8.5		19.6	MHz
$f_{\text{DCO}(7,31)}$	DCO frequency (7, 31)	DCORSELx = 7, DCOx = 31, MODx = 0	60		135	MHz
S_{DCORSEL}	Frequency step between range DCORSEL and DCORSEL + 1	$S_{\text{RSEL}} = f_{\text{DCO}(\text{DCORSEL}+1, \text{DCO})} / f_{\text{DCO}(\text{DCORSEL}, \text{DCO})}$	1.2		2.3	ratio
S_{DCO}	Frequency step between tap DCO and DCO + 1	$S_{\text{DCO}} = f_{\text{DCO}(\text{DCORSEL}, \text{DCO}+1)} / f_{\text{DCO}(\text{DCORSEL}, \text{DCO})}$	1.02		1.12	ratio
	Duty cycle	Measured at SMCLK	40	50	60	%
df_{DCO}/dT	DCO frequency temperature drift ⁽¹⁾	$f_{\text{DCO}} = 1 \text{ MHz}$		0.1		%/°C
$df_{\text{DCO}}/dV_{\text{CC}}$	DCO frequency voltage drift ⁽²⁾	$f_{\text{DCO}} = 1 \text{ MHz}$		1.9		

/*通过 DCO-FLL 将32.768KHZ 倍频到25MHZ*/

#include <msp430.h>

#include "HAL_PMM.h"

void delay()

{

volatile unsigned int I;

for(I = 0; I != 5000; ++i){_NOP(); }}//延时函数

void main(void) {

WDTCTL = WDTPW+WDTHOLD;

P1SEL &= ~BIT1;

```

P1DIR |= BIT1;

P1SEL |= BIT0; //ACLK
P1DIR |= BIT0;
P2SEL |= BIT2; //SMCLK
P2DIR |= BIT2;
P7SEL |= BIT7; //MCLK
P7DIR |= BIT7;
P5SEL |= BIT4|BIT5;
UCSCTL6 |= XCAP_3;
UCSCTL6 &= ~XT10FF; //打开 XT1, 否则 XT1LFOFFG 可能报错
SetVCore(3); //提高 Vcore 电压到最高级, 以满足倍频需求该函数位于
HAL_PMM.H 中
__bis_SR_register(SCG0); //该语法为固定格式, 意为将括号内的变量
置位, SCG0 与系统工作模式有关, 此时 MCLK 暂停工作
UCSCTL0 = 0; //先清零, FLL 运行时, 该寄存器系统会自动配置, 不
用管
UCSCTL1 = DCORSEL_6;
UCSCTL2 = FLLD_1 | 380; //FLLD=1, FLLN=380, 则频率为
                        2*(380+1)*32.768=24.969MHZ
__bic_SR_register(SCG0);
__delay_cycles(782000); //系统自带的精确延时, 单位 us
while(SFRIFG1 & OFIFG) {
    UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
    SFRIFG1 &= ~OFIFG;
}

UCSCTL4 = UCSCTL4 & ~(SELS_7|SELM_7) | SELS_3|SELM_3;

//选择 DCO 作为时钟源

while(1){

P1OUT ^= BIT1;
delay();
}
}

```

MSP430F5529 （四）电源&&& （五）工作模式

电源管理模块（PMM）和供电监控简介 Power Management Module and Supply Voltage Supervisor

我觉得电源管理与监控是一个很复杂很难掌控的部分，不仅涉及到到源模式的选择，还牵扯到复杂的中断、以及中断如何处理等等。虽然学好这一部分对实现降低功耗的目的很有帮助，但对于目前的我们来说貌似““功耗”一词还稍微远了点。此外，这部分控制对防止和处理供电意外（（过高过低等）的发生很有帮助，不过貌似这个开发板如果不独立拿来做个项目而只是接在电脑 USB 供电的话，一般也不会有什么问题的。所以，我也只打算简单学一下。（到后期有机会会再仔细学）。

I/O 口和所有模拟单元包括晶振在内都由 DVCC 供电。内存(flash 和 RAM)和数字单元由核心电压 VCore 供电。

DVCC: 宽的电源电压范围1.8V-3.6V;

VCore: DVCC 经低压降电压调整器(LDO), 产生的一个二次核心电压，专门为 CPU 数字逻辑供电，共有1.4V (0-12MHz) ,1.6V (0-16MHz) ,1.8V (0-20MHz) 和1.9V (0-25MHz) 四个级别。VCore 的最小允许电压依赖于选择的 MCLK 大小，也就是说高主频时需要配套较高的 VCore。

管理会产生复位（主要是上电期间），监控会产生中断（监控电压是否过高过低）。

我们最常用的是设置核心电压 VCore，还好有专门的函数库 HAL_PMM.c/h。在这个函数库里除了一些设置的定义外，最重要的就是定义了三个函数：

SetVCoreDown(uint8_t level):降低核心电压

SetVCoreUp(uint8_t level): 提高核心电压

SetVCore(uint8_t level):直接设置核心电压值（0-3共四级）/*这个函数最重要，或者说有了这一个，前面两个就不需要了*/

五、系统工作模式

第四章我们提到可以从电源层入手，达到从源头上控制功耗的目的。这一章 我们就会讲 CPU 工作模式，如何从次一级来控制功耗。

不同工作模式下，CPU 会禁用一些模块，从而达到控制功耗的目的。

（PS：同样那句话，“功耗太远”，一般都不会去更改工作模式，所以简单学习）

简介几句话：①改变工作模式会立即生效；

②发生中断时，当前的模式设置信息会被保存，以便恢复（除非中断服务程序中改变了工作模式）；

③处于 LPM4.5模式时，PMM 的电源监管不会生效，所有的 RAM 和寄存器都会丢失，但是 I/O 口状态会锁定；

④从 LPM4.5唤醒，有一套专门的流程，有兴趣就去看；

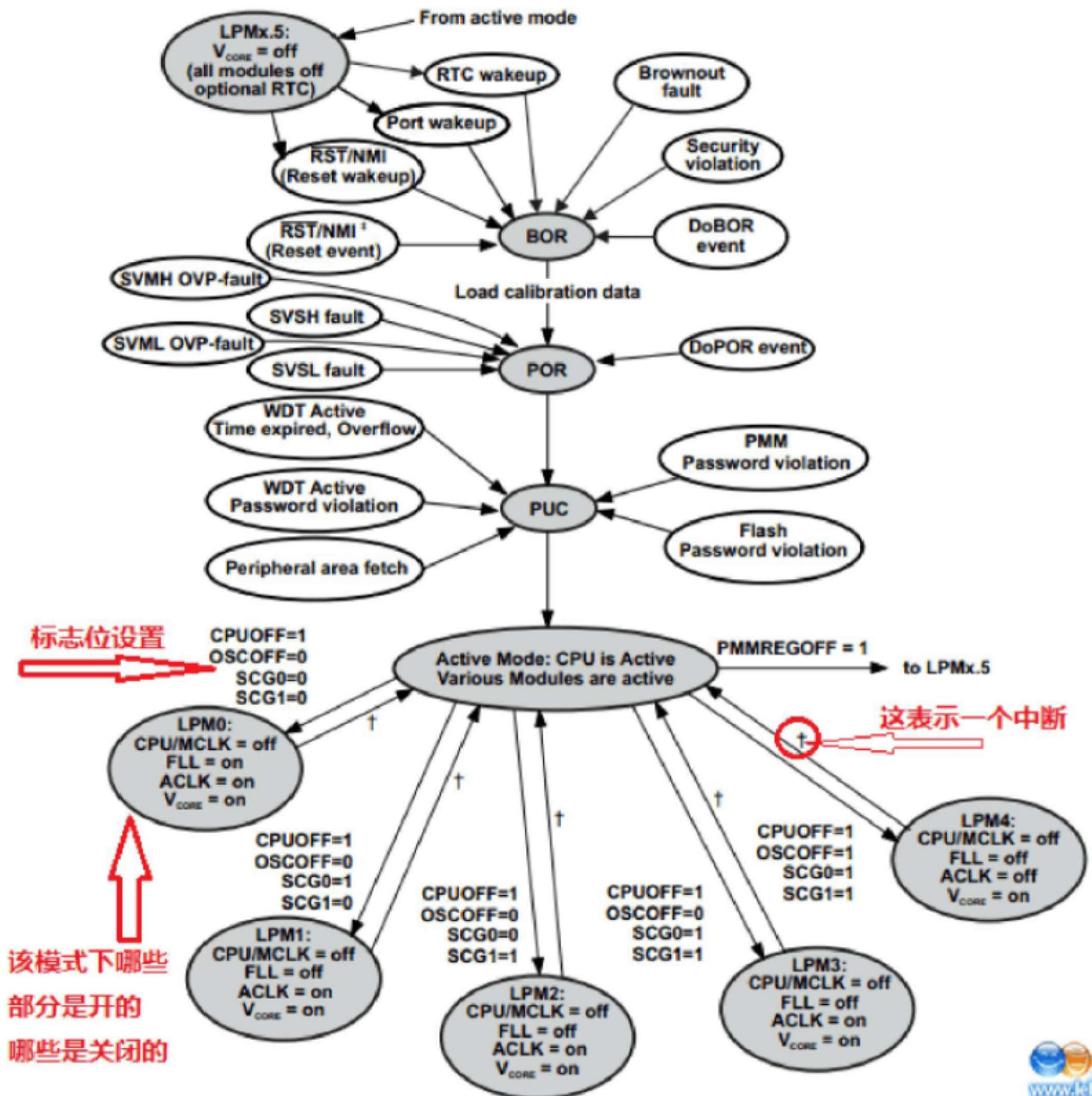
下面这张图很有意思，可以清楚的看清工作模式之间转换的流程与方向，以及每种工作模式是怎样设置的，又控制了哪些部分。

BOR: Brown-Out Reset 低电压检测复位（欠压复位）

POR: Power-On Reset 上电复位

PUC: Power-Up Clear 上电清除

浅色部分表示一个事件，深色部分表示一种操作或设置



- ①设置工作模式主要是设置寄存器 SR 的 SCG0、SCG1、OSCOFF、CPUOFF 位，AM (Active Mode) 模式时四位均置零，且系统默认为 AM 模式；
- ②除了 AM, 其余都为低功耗模式，处理器进入低功耗模式以后，一般由中断来唤醒。可以是外部中断，也可以是内部的定时器等中断；
- ③LPM0-LPM4 模式下，外围模块都会正常工作，且 RTC 时钟不会停止；
- ④要进入 LPM4.5 这一模式 (更少用)，只需在 LPM4 的基础上多一个 PMMREGOFF 置位。该模式下，系统的所有时钟、内存和监督管理机制都停了，连实时时钟 RTC 都禁止操作了。
- ⑤LPM0 和 LPM1 一组，除了上图显示的特征外，该模式下 SMCLK 是选通的 (SMCLKOFF = 0)，DCO 的时钟源如果是 ACLK 或者 SMCLK，则 DCO 也是有效的；
- ⑥ LPM2 和 LPM3 一组，除了上图显示的特征外，该模式下 SMCLK 是禁止的，DCO 的时钟源如果是 ACLK，则 DCO 也是有效的；

⑦MSP430的头文件对低功耗模式有详尽的定义，如：要进入低功耗模式0，可在程序中直接写 LPM0；进入低功耗模式4，可以直接写 LPM4。退出低功耗模式如下：

LPM0_EXIT; //退出低功耗模式0 //太方便了有木有

LPM4_EXIT; //退出低功耗模式4 (LPM4.5除外)

总结实验：一个很有意思的程序

/*低功耗模式的体现，不用无限循环，程序也不会终止*/

/*大家会发现，LPM3后面的语句不会执行，程序只会定期执行一下中断服务程序，这是因为LPM3模式下MCLK和SMCLK、FLL都禁止了*/

#include <msp430.h>

void main(void)

{

WDTCTL=WDTPW+WDTCNTCL+WDTTMSSEL+WDTIS1+WDTIS0; //WDT作定时器用

SFRIE1|=WDTIE; //开看门狗中断

P1DIR|=BIT1+BIT2; //P1.1接LED，设定为输出方向

P1OUT=BIT1+BIT2;

__enable_interrupt(); //开总中断

//_BIS_SR(GIE); 这句话的意思也是开总中断

/*这里掌握一个用法_BIS_SR()：将括号内的变量置位*/

LPM3; //进入LPM3低功耗模式，此模式下SMCLK被禁止

P1OUT&=~BIT2; //这句话执行不了，所以P1.2就会保持常亮，而不会变暗

}

#pragma vector=WDT_VECTOR

__interrupt void WatchTimer(void)

{

P1OUT^=BIT1; //定时翻转，以实现闪烁

}

MSP430F5529 （六）定时器 Timer_A-1

定时器 Timer_A

MSP430F5529共有两类共4个定时器，分别是 Timer_A 定时器3个和 Timer_B 定时器1个，按照每个寄存器配备的捕获/比较器的个数分别命名为 Timer0_A（内有5个捕获比较器）、Timer1_A（3个）、Timer2_A（3个）、Timer0_B（7个）。

这一章，我们讲定时器 Timer0_A。（A类的都一样）

注意：下面所提到的所有寄存器，在TA后面插入0或1或2就分别表示 Timer0_A、Timer1_A、Timer2_A（我这里省略了数字）

6.1 简介一下

定时器A是一个复合了捕获/比较寄存器的十六位的定时（加减）计数器。定时器A

支持多重捕获/比较, PWM 输出和内部定时, 具有扩展中断功能, 中断可以由定时器溢出产生或由捕获/比较寄存器产生。

特征简介:

○四种运行模式的异步16位定时/计数器

○自身时钟源可选择配置

○最多达5个可配置的捕获/比较寄存器 (CCR)

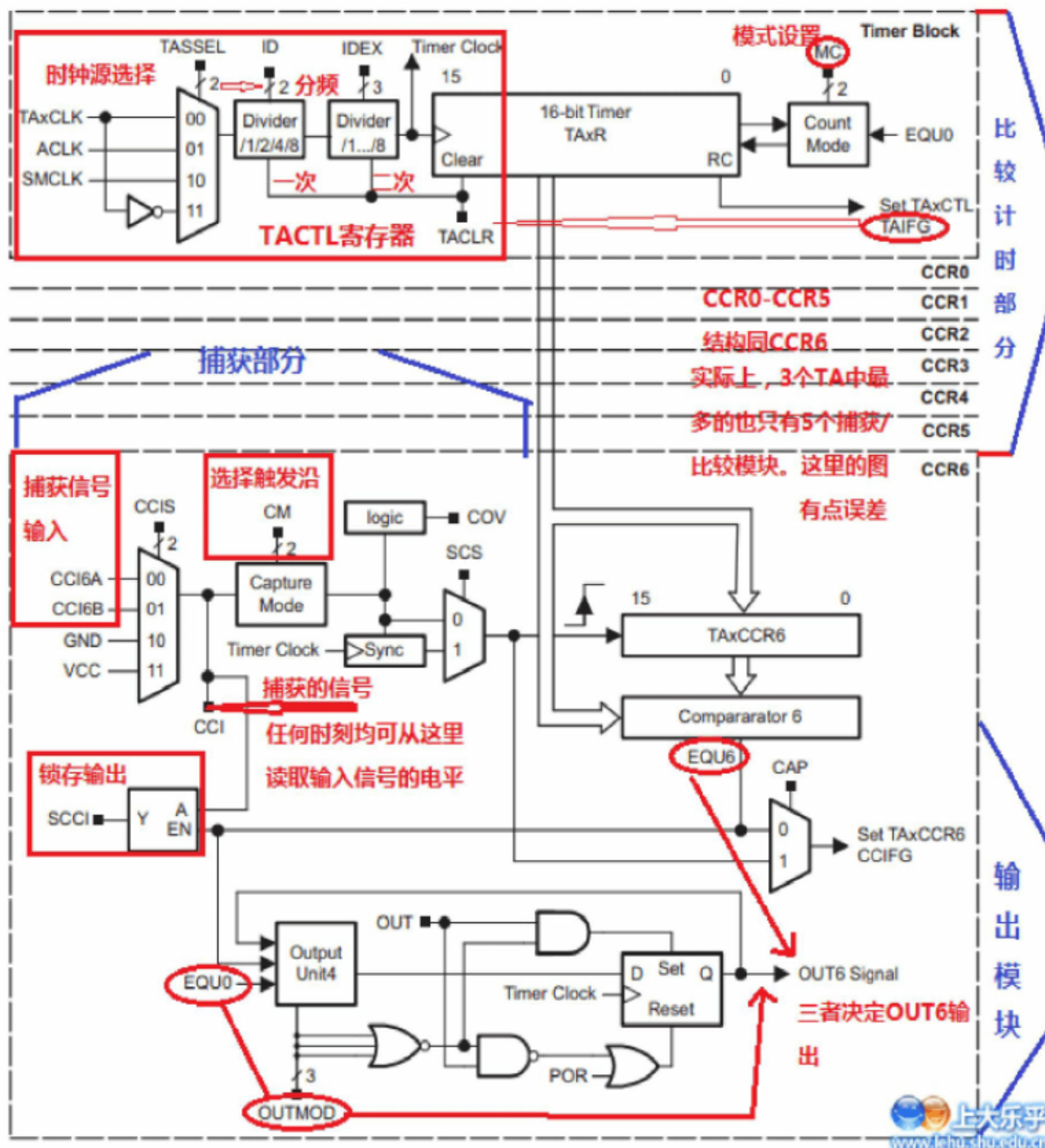
capture/compare registers

○可配置的 PWM 输出

○异步输入和输出锁存

○对所有 Timer_A 中断快速响应的中断向量寄存器

下面这张图形象的解释了 Timer_A 的结构特性



6.2 TA (Timer_A) 的几个基本操作设置 (含寄存器介绍及设置)

声明：所有寄存器同样支持字和字节操作，不要忘记这是什么意思

所有寄存器初始化都为0x0000

6.2.1 TA 控制寄存器 TACTL (最常用最基本)

再次说明一下例如：TA0CTL、TA1CTL、TA2CTL 分别表示3个不同定时器 A 的控制寄存器

rw-(0)表示默认读写均为0

15	14	13	12	11	10	9	8
Unused						TASSEL	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ID		MC		Unused	TACLRL	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

TASSELx：时钟源选择。尽量不要选 TASSEL0-TACLK 外部时钟源，因为如果 TACLK 和 CPU 时钟不同步，很容易出问题。(TA0CLK 接 P1.0 引脚)

- 00 TACLK
- 01 ACLK
- 10 SMCLK
- 11 ~TACLK

IDx：第一次分频控制。ID0-1分频；ID1-2分频；ID2-4分频；ID3-8分频
MC：工作模式控制。(建议在修改定时器运行模式前先停止定时器(中断使能、中断标志、TACLRL 例外)，以避免产生未知的误操作。)

- 00 停止模式：定时器停止
- 01 增模式：定时器计数到 TACCR0
- 10 连续模式，定时器计数到0FFFH
- 11 增减模式：定时器加计数到 TACCR0然后减计数到0000H

TACLRL：定时器清零位。该位置位会复位 TA 寄存器，时钟分频和计数方向。
TACLRL 位会自动复位并置0

TAIE：定时器中断使能

- 0：中断禁止
- 1：中断允许

TAIFG：中断标志位

- 0：没有中断发生
- 1：有中断挂起

6.2.2 计数值存放寄存器 TAR

- ①显然，最大存放计数值为0xFFFFh；
- ②(类似51单片机)可以被用来存放一个初值，然后选用连续模式。这样不断计满再手动填充，从而达到精确计时的效果；

③默认为0，且对该寄存器可以直接赋值；

6.2.3 扩展寄存器 **TAEX0**

很简单，这个寄存器就是为了控制时钟源的二次分频（看结构图）。
该寄存器的低3为定义为 **TAIDEX**：000-111分别表示1-8分频

6.2.4 捕获/比较寄存器 **TACCR0-TACCR4**（共5个）

比较模式下，用来设定计数终值；
捕获模式下用来将捕获的 **TAR** 值存放在 **TACCRx** 中。

6.3 MC 控制的四种工作模式的详细讲解

6.3.1 MC=0停止模式

这是系统默认的模式，定时计数器禁止工作。

6.3.2 MC=1增模式

总结几句话：（红色标记的很重要）

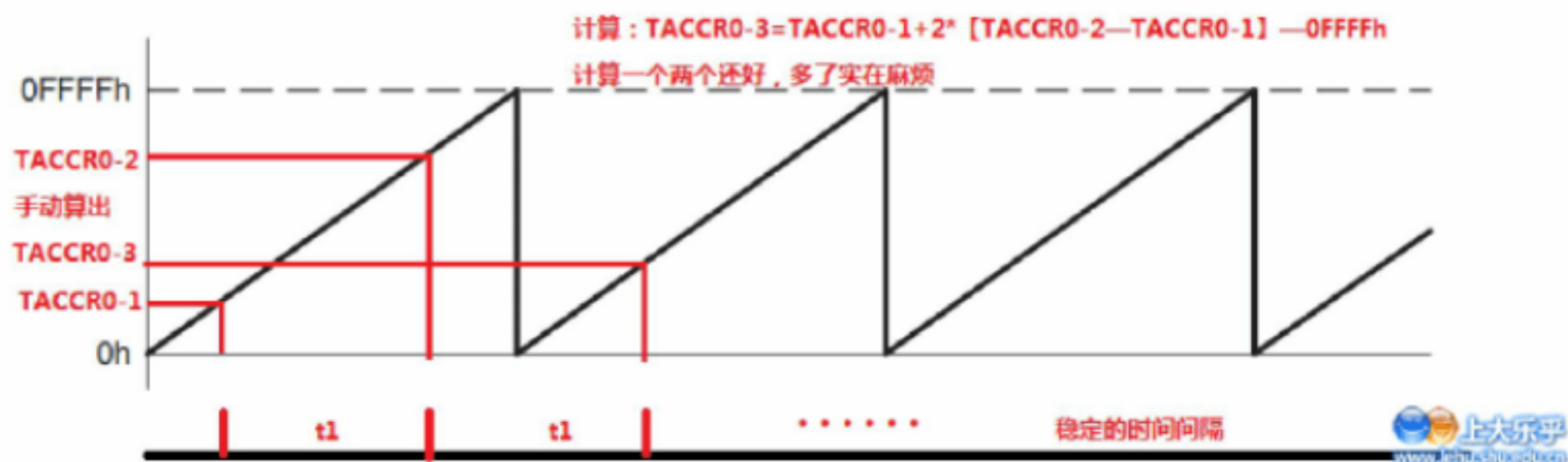
- ①此模式下严禁从0xffff开始计数；
- ②注意从0计到 **TACCR0**，实际上记了 **TACCR0+1**个数；
- ③计到 **TACCR0**后，会回到0重新开始计数；
- ④如果 **TAR** 的值大于 **TACCR0**，这时候会立即从0开始计数；
- ⑤当定时器计数到 **TACCR0**的值时，**中断标志 CCIFG 位**（之后会讲到）置位。当定时器由 **TACCR0**返回0时，**TAIFG 中断标志置位**；
- ⑥在定时器运行时修改 **TACCR0**，如果新的周期值大于或等于旧的周期值，或大于当前的定时器计数值，那么定时器立刻开始执行新周期计数。如果新周期小于当前的计数值，那么定时器回到0。但是，**在回到0之前会多一个额外的计数。**

6.3.3 MC=2连续模式

在连续模式中，定时器重复计数到0FFFFH，然后重新从0开始增计数（除非每次重装计数初值）。当定时器从0FFFFH到0时，**TAIFG 中断标志置位**。

应用：连续模式下利用捕获/比较器产生需要的时间间隔。原理是：计数一直在进行，捕获器 **TACCRX** 中存有第一个计数终值，每次捕获器计到 **TACCRX** 时，会产生中断标志，我们可以在中断服务函数中写入一个计算好的下一个的计数终值，这样无限计算和中断下去，那么该捕获器就会产生一个稳定的时间间隔序列。（其实吧，不明白也没关系。就算明白了，也不好，因为计算起来很麻烦而且也不好用）

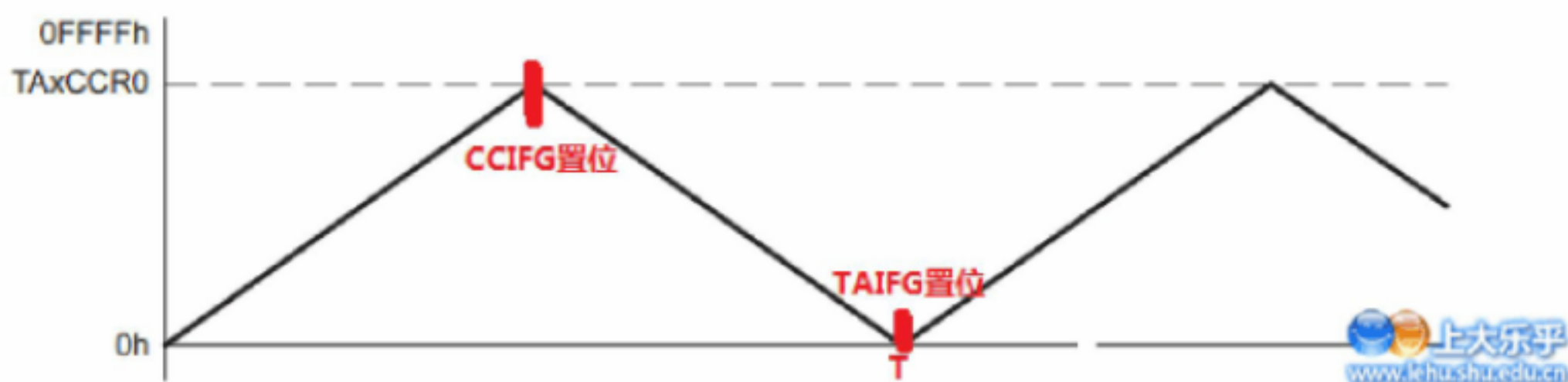
如图：



6.3.4 MC=3增减模式（常用于生成 PWM 波）

①该模式下，计数方向是固定的，即让定时器停止后再重新启动定时器，它就沿着停止时的计数方向和数值开始计数。如果不希望这样，就需要将 TACLRL 置位来清除方向。TACLRL 位也会清除 TAR 的值和定时器的时钟分频。

②此模式下置位情况如下图：



③当定时器运行时，改变 TACCR0 的值，如果正处于减计数的情况，定时器会继续减到 0，新的周期在减到 0 后开始。

如果正处于增计数状态，新周期大于等于原来的周期，或比当前计数值要大，定时器会增计数到新的周期；如果新周期小于原来的周期，定时器立刻开始减计数，但是，在定时器开始减计数之前会多计一个数。

MSP430F5529 （六）定时器 Timer_A-2

6.4 捕获比较模块

这是在以上介绍的基础上正式讲 TA 的重要功能。

先看一个寄存器 TACCTL0-TACCTL6：（TA 中最复杂的寄存器，用到的时候查表啦）

15	14	13	12	11	10	9	8
CM	CCIS	SCS	SCCI	Unused	CAP		
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)
7	6	5	4	3	2	1	0
OUTMOD	CCIE	CCI	OUT	COV	CCIFG		
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	

CMx: 捕获模式设定 **00** 不捕获
01 上升沿捕获
10 下降沿捕获
11 上升和下降沿都捕获

CCISx: 捕获源的选择 **00** CCIxA
 01 CCIxB
 10 GND
 11 VCC

SCS: 同步捕获源，设定是否与时钟同步
 0 异步捕获
 1 同步捕获

SCCI: 选择的 CCI 输入信号由 EQUx 信号锁存，并可通过该位读取。

CAP: **0**-比较模式 **1**-捕获模式

OUTMOD: 输出模式控制位。（之后会在输出模块详细解释）

CCIE: 中断使能，该位允许相应的 CCIFG 标志中断请求。

0-中断禁止 **1**-中断允许

CCI 3 : 捕获比较输入，所选择的输入信号可以通过该位读取

OUT : 对于输出模式**0**，该位直接控制输出状态。

0-输出低电平 **1**-输出高电平

COV: 捕获溢出位。该位表示一个捕获溢出发出，COV 必须由软件复位。

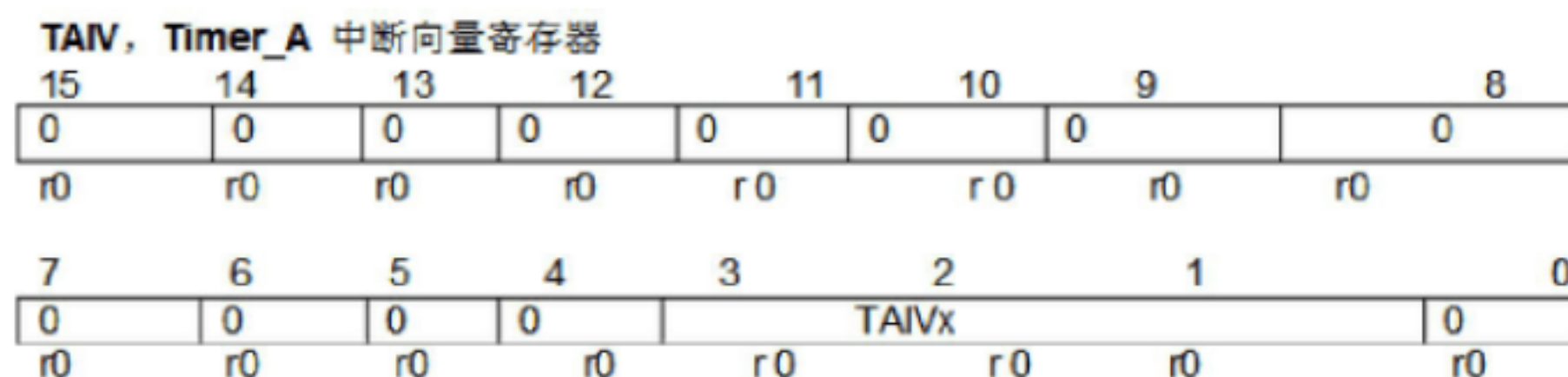
0-没有捕获溢出发生 **1**-有捕获溢出发生

CCIFG: 捕获比较中断标志位。

0-没有中断挂起 **1**-有中断挂起

最后一个寄存器 TAIV:（还记得外部中断寄存器吗，里面同样存储的只是一个中断代号）

里面没有 **TACCR0**的中断标志，因为 **TACCR0**优先级最高，有一个专门的中断向量）



TAIVx Bits 15-0 Timer_A 中断向量值

TAIV 的内容	中断源	中断标志	中断优先级
00H	无中断挂起	——	
02H	捕获比较 1	TACCR1 CCIFG	最高
04H	捕获比较 2	TACCR2 CCIFG	
06H	捕获比较 3	TACCR3 CCIFG	
08H	捕获比较 4	TACCR4 CCIFG	
0AH	捕获比较 5	TACCR5 CCIFG	
0CH	捕获比较 6	TACCR6 CCIFG	
0EH	定时器溢出	TAIFG	最低位

这里面的标志位需要软件手动清零。一种情况例外：两个中断同时发生，先响应优先级高的中断，当该中断服务程序结束后，该位的中断标志会自动清零，然后去响应另外一个中断。

6.4.1 比较模式

TA 启动时默认为比较模式。
(CAP=0时选择比较模式)

比较模式简介：（也就是一般意义上的定时计时模式）

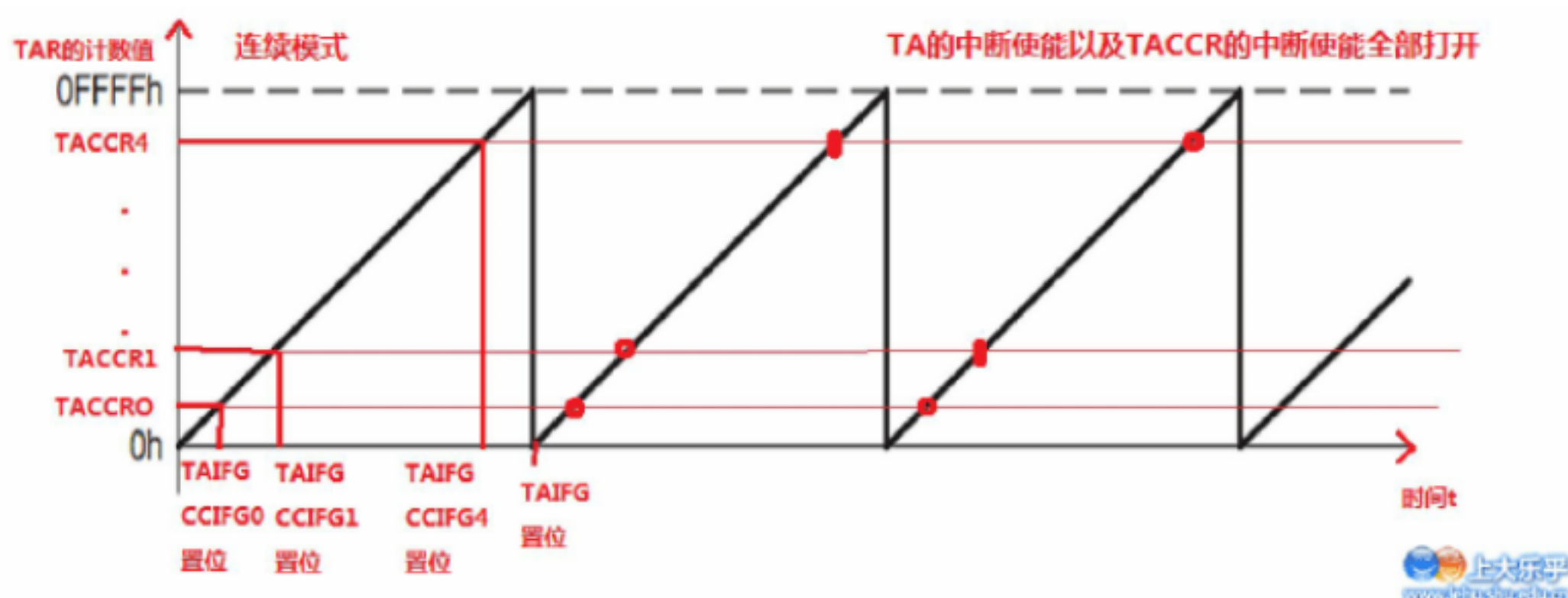
这是定时器的默认模式，当在比较模式下的时候，与捕获模式相关的硬件停止工作，如果这个时候开启定时器中断，然后设置定时器终值(将终值写入 TACCRx)，开启定时器，当 TAR 的值增加到和某个 TACCRx 里面的值相等的时候，相应的中断标志位 CCIFGx 置一，同时中断标志位 TAIFG 置位。**若中断允许未开启则只将中断标志位 CCIFGx 置一。**

（还记得51单片机的定时器吗）

注意：当 Timer_A 要用到 TACCR0 的值作为终值来计数（也就是增模式或者增减模式），很显然 TACCR0 的值一定要大于其 TACCRx 的值，否则那些比 TACCR0 大的计数值就没有存在的意义了。

下面是我画的一个图。比较形象的解释了工作原理。（期间 TACCR 的值不改变）

所谓的比较就是，如果计数器 TAR 中的值和某个 TACCRx 中的值相等了，那么相应的标志位就会置位。



这只是一个原理，实际应用的时候，会很灵活，通过一个一个设定每次的 TACCR 值，可以得到想要的各种时间间隔。

总结：比较模式用于选择 PWM 输出信号或在特定的时间间隔中断。当 TAR 计数到 TACCRx 的值时：

- 相应的中断标志 CCIFG 置位；
- 内部信号 EQUx=1
 - EQUx 根据输出模式来影响输出信号
- 输入信号 CCI 锁存到 SCCI

6.4.2 捕获模式

当 CAP=1时，选择捕获模式。捕获模式用于记录时间事件，比如速度估计或时间测量。捕获输入 CCIXA 和 CCLXB 连接外部的引脚或内部的信号，这通过 CCISX 位来选择。CMX 位选择捕获输入信号触发沿：上升沿、下降沿或两者都捕获。当输入信号的触发沿到来时，捕获事件发生：

- 定时器的 TAR 值复制到 TACCRX 寄存器中
- 中断标志位 CCIFG 置位

注意：①捕获信号可能会和定时器时钟不同步，并导致竞争条件的发生。将 SCS 位置位可以在下个定时器时钟使捕获同步

②如果第二次捕获发生时，第一次捕获的 TAR 值还没有及时被存到 TACCRx，捕获比较寄存器就会产生一个溢出逻辑，COV 位在此时置位，COV 位必须软件清除。

6.5 输出模块

传统的定时器，都是通过标志位的判断来定时触发事件的。而430则具有输出模块，通过和定时结合起来，可以方便的产生 PWM 信号或者其它控制信号

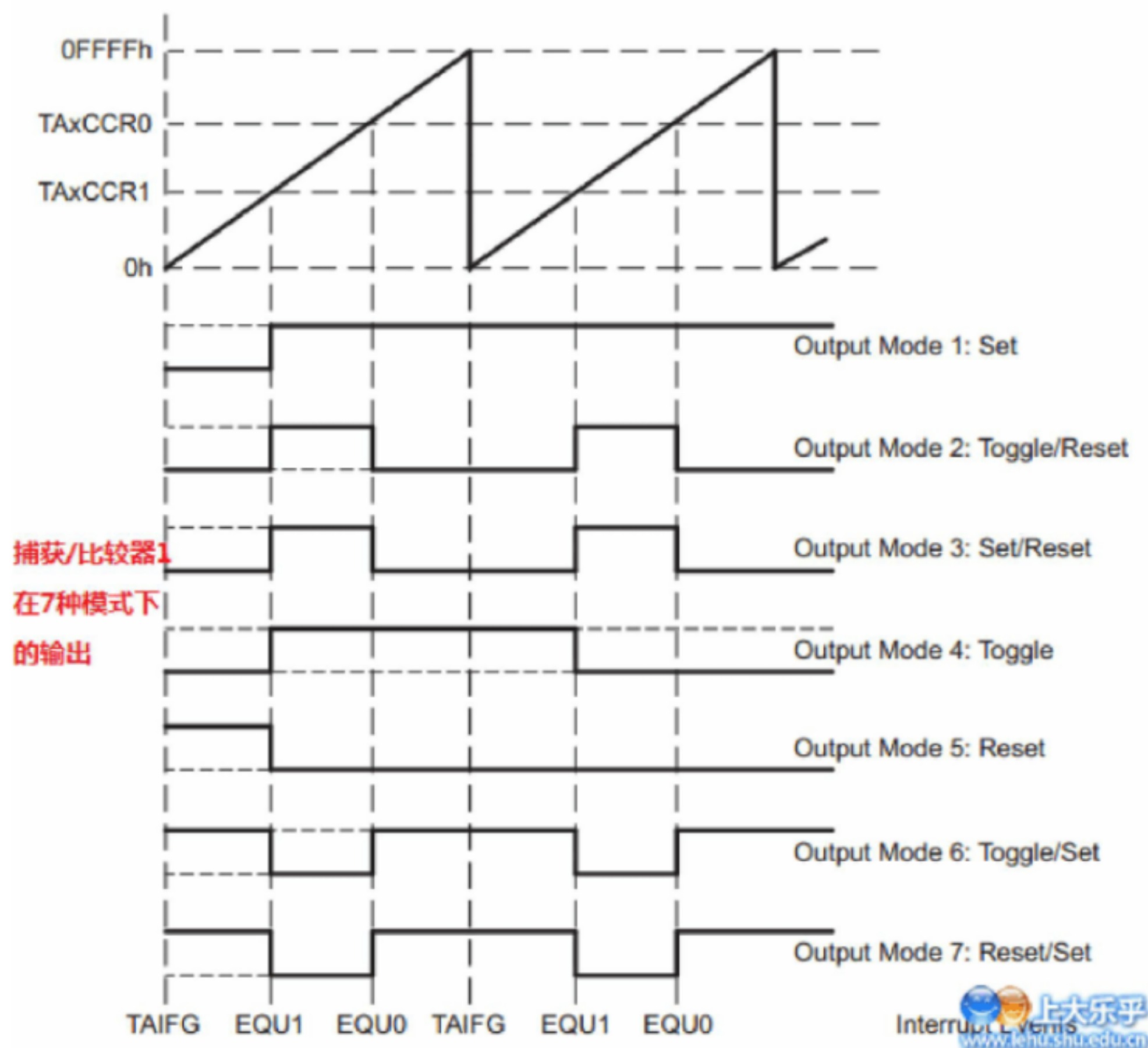
每个捕获/比较器都有一个输出口，如 P1.1-P1.5对应 TA0.0-TA0.4这5个捕获比较器的输出。

输出模式：输出模式由 OUTMODx 位来确定，如下表对于所有模式来说（模式0除外），OUTx 信号随着定时器时钟的上升沿而改变。输出模式2，3，6和7对输出模式0无效，因为此模式下，EQUx=EQU0。

（复位指的是置0）

OUTMODX	模式	说明
000	输出	输出信号 OUTx 由 OUT 位定义。当 OUT 位更新时，OUTx 信号立刻更新
001	置位	当定时器计数到 TACCRX 值时，输出置位，并保持置位直到定时器复位或选择了另一个输出模式
010	翻转/复位	当定时器计数到 TACCRX 值时，输出翻转。当定时器计数到 TACCRO 值时，输出复位
011	置位/复位	当定时器计数到 TACCRX 值时，输出置位。当定时器计数到 TACCRO 值时，输出复位
100	翻转	当定时器计数到 TACCRX 值时，输出翻转。输出信号的周期将是定时器的2倍
101	复位	当定时器计数到 TACCRX 值时，输出复位，并保持复位直到选择了另一个输出模式
110	翻转/置位	当定时器计数到 TACCRX 值时，输出翻转。当定时器计数到 TACCRO 值时，输出置位
111	复位/置位	当定时器计数到 TACCRX 值时，输出复位。当定时器计数到 TACCRO 值时，输出置位

举一个例子：结合上表看



注意：在模式转换的时候，一定要保持 OUTMOD 至少一位置位，除非转向 0 模式。所以最好的做法是：先把 OUTMOD 置为 7，然后再清除掉不需要的位。

做一个说明：比较模式下，当计数器 TAR 中的值和 TACCRX 中的设计值相等时，相应捕获/比较器的 EQUx 就会置位。那么 EQU0、EQUx 和 OUTMOD 是怎么来影响输出的呢？以模式 2（翻转/复位）为例，该模式的定义是这样的：当定时器计数到 TACCRX 值时，输出翻转。当定时器计数到 TACCR0 值时，输出复位。于是，这句话也可以翻译成在模式 2 的条件下，当 EQUx=1 时，输出翻转；当 EQU0 等于 1 的时候，输出复位。这两个信号这里相当于两个触发（使能）信号了。

总结

实验一：

/*利用 Timer_A 比较模式下的多路定时，让 LED 闪烁*/

#include <msp430f5529.h>

void main(void)

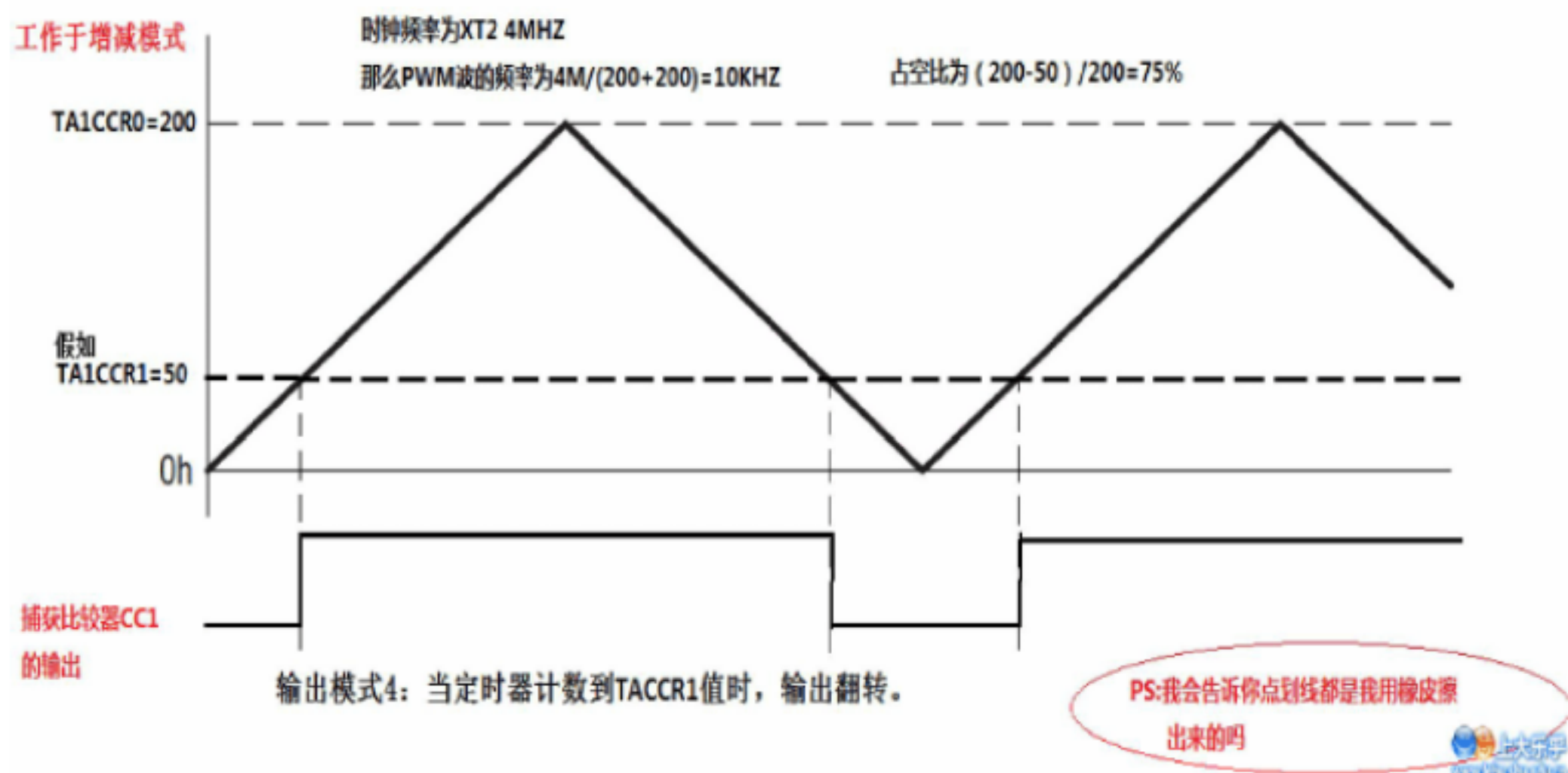
```

{
    WDTCTL=WDTPW+WDTHOLD;
    P1DIR|=(BIT1+BIT2+BIT3+BIT4+BIT5); //P1.1-P1.5为输出方向
    P1OUT=0x00; //全部拉低，初始化 LED 全灭
    TA0CCTL1=CCIE; //捕获比较器1开启 CCIFG 位中断
    TA0CCR1=13107; //置入要比较的数值0xff/5=13107
    TA0CCTL2=CCIE; //捕获比较器2开启中断
    TA0CCR2=26214; //13107*2=26214
    TA0CCTL3=CCIE; //捕获比较器3开启中断
    TA0CCR3=39321; //13107*3=39321
    TA0CCTL4=CCIE; //捕获比较器4开启中断
    TA0CCR4=52428; //13107*4=52428
    TA0CTL|=TACLRL+TAIE; //开启中断并清零
    TA0CTL|=TASSEL_1+MC_2+TAIE; //选择 SCLK32.768KHZ 作为时钟，
    选用连续模式，并开启中断
    /*这样的话，5个灯闪一遍的时间为0xffff/32768=2S*/
    __enable_interrupt(); //开启总中断
    while(1);
}

/*TIMER0_A0_VECTOR 是计时器0的 CCR0的中断寄存器，
    TIMER0_A1_VECTOR 是计时器0的 CCR1-CCR4、TA 的寄存器*/
/*同理定时器 TA1也是分为两个 TIMER1_A0_VECTOR 和
    TIMER1_A1_VECTOR*/
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TimerA(void)
{
    switch(__even_in_range(TA0IV,14))
    /* 这句话的意思是：只有在 TA0IV 的值是在0--14内的偶数时才会执行 switch 函数内的语句
        其作用是提高 switch 语句的效率*/
    {
        case 2:P1OUT=BIT1;break; //TACCR1 CCIFG 置位，表明计数
        值和设定的13107相等了，也就是说计了0.4S 了
        case 4:P1OUT=BIT2;break; //TACCR2 CCIFG 置位，表明计了
        0.8S 了
        case 6:P1OUT=BIT3;break; //TACCR3 CCIFG 置位，表明计了
        1.2S 了
        case 8:P1OUT=BIT4;break; //TACCR4 CCIFG 置位，表明计了
        1.6S 了
        case 14:P1OUT=BIT5;break; //TAIFG 置位，表明计了2S 了
        default:break;
    }
}
}

```


实验二：比较模式-增减模式输出 PWM 波



/*在比较和增减模式下产生 PWM 波（矩形波）*/

/*提一个 PWM 波的用处：驱动直流电机。我们知道对于直流电机，驱动它的
电流的频率并不影响转速，只有占空比会影响转速*/

/*开发板上 P2.0是有外接排针的，所以用这一端口输出 PWM*/

/*看 CPU 引脚发现，P2.0为 TA1.1，也就是定时器 A1的1号捕获比较器输
出口*/

```
#include <msp430.h>
```

```
void main(void)
```

```
{
```

```
    WDTCTL=WDTPW+WDTHOLD;
```

```
    P2SEL |= BIT0; //声明有特殊功能，不做普通 I/O 使用
```

```
    P2DIR |= BIT0; //输出
```

```
    P2DS |= BIT0; //全力驱动，否则可能无法驱动电机
```

```
    P2OUT&=~BIT0; //初始化输出低电平
```

```
    /*把 SMCLK 配置为 XT2 4MHZ*/
```

```
    P5SEL=BIT2+BIT3; //声明特殊功能，将用作外部时钟晶振 XT2输入
```

```
    UCCTL6&=~XT2OFF; //开启 XT2
```

```
    while(SFRIFG1 & OFIFG)
```

```
    {
```

```
        UCCTL7 &=~(XT2OFFG+DCOFFG+XT1LFOFFG); //清除3种时钟错误  
        标志
```

```
        SFRIFG1&=~(OFIFG); //清除时钟错误标志位
```

```
    } //直到 XT2从起振到振荡正常，没有错误发生
```

```
    UCCTL4|=SELS_5; //把 SMCLK 的时钟源选为 XT2 4MHZ
```

```
    TA1CTL0=CCIE; //定时器 A1的捕获比较器0开启 CCIFG 位中断
```

```
    TA1CCR0=200; //置入计数终值，则 PWM 频率为10KHZ
```

```

    TA1CCTL1=CCIE;      //捕获比较器1开启中断
    TA1CCR1=50;         //占空比为75%
    TA1CTL|=TACLK;      //将计时器 A1清零
    TA1CTL|=TASSEL_2+MC_3; //定时器选择 SMCLK 作为时钟源，且为
    增减模式
    TA1CCTL1=OUTMOD_4; //定时器 A1中的捕获比较器1，输出模式为4翻
    转
    while(1);
}

//呼吸灯//
// 介绍：该程序利用 TIMER A 的 UP 模式 在 P1.3脚产生 PWM 输出
// 将 CCR0设置为1500来定义 PWM 的周期，利用循环不断改变 CCR1的值，
// 实现利用改变 PWM 的占空比来改变 LED 亮度。
// SMCLK = MCLK = TACLK = default DCO
#include <msp430f5529.h>
void delay_nms(unsigned int n)// 延时函数
{
    unsigned int j;
    for (j=0;j<(n);j++)
    {
        __delay_cycles(400); //太短会使 LED 显得好像在常亮，太长就要等
        较长时间来观察了
    }
}
void main(void)
{
    unsigned const PWMPeriod = 1500; //设置 PWM 周期参数，const 声
    明此值不允许改变.该数值太大，会导致 LED 闪烁
    volatile unsigned int i; //声明变量 i 是随时可变的，系统不要去优
    化这个值
    WDTCTL = WDTPW + WDTHOLD; // 关闭看门狗
    P1DIR |=BIT3; // 设置 P1.3为输出
    P1SEL |=BIT3; // 设置 P1.3为 TA0.2输出
    TA0CCR0 = PWMPeriod; // 设置 PWM 周期
    TA0CCTL2 = OUTMOD_7; // 设置 PWM 输出模式为：7 - PWM 复位/置位模
    式，
    // 即输出电平在 TAR 的值等于 CCR2时复位为0，当 TAR 的值等于 CCR0时
    置位为1，改变 CCR2，从而产生 PWM。其实模式2也可以
    TA0CTL= TASSEL_2 +MC_1; // 设置 TIMERA 的时钟源为 SMCLK，计数模
    式为 up,到 CCR0再自动从0开始计数
    while(1)
    {
        TA0CCR2=0;//确保最开始是暗的
        //渐亮过程：不断设置 TA0CCR2的值，使翻转的时间变长，改变 PWM 的占

```



```

        空比
for(i=0;i<PWMPeriod;i+=1)
{
    TA0CCR2=i;
    delay_nms(4-(i/500)); // 占空比变化的延时，调整延迟时间可改变呼吸
        灯变暗的速度
    //在暗的时候延长 delay 时间，可增强效果
}
//渐暗过程：不断设置 TA0CCR2的值，使翻转的时间变短，改变 PWM 的占
    空比
for(i=PWMPeriod;i>0;i-=1)
{
    TA0CCR2=i;
    delay_nms(4-(i/500)); // 占空比变化的延时，调整延迟时间可改变呼吸
        灯变暗的速度
    //在暗的时候延长 delay 时间，可增强效果
}
TA0CCR2=0; //确保灯暗
delay_nms(250); //时间长一点，增强视觉效果
}
}

```

七、定时器 Timer_B

定时器 B 和定时器 A 有很多相同之处，学习的时候注意回忆 Timer_A 的 相关知识。
注意，MSP430F5529中只有一个定时器 B。

7.1 定时器 B 的简介

7.1.1特性（了解）：

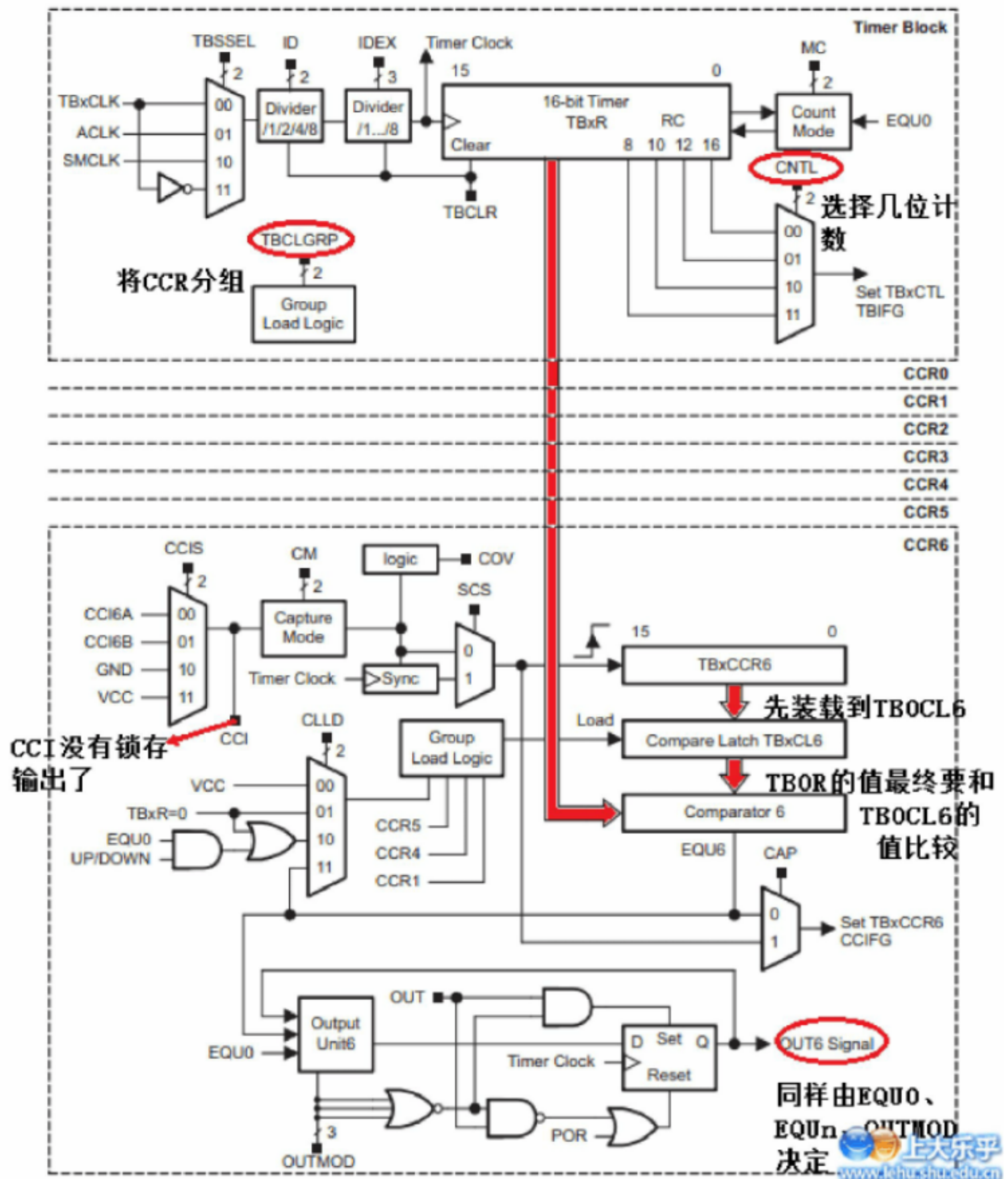
- (1). 16位同步定时/计数，4种工作模式可选、4中长度可选；
- (2). 可选可配置时钟源；
- (3). 高达7个捕获/比较寄存器；
- (4). 可配置 PWM 输出；
- (5). 带有同步装载的双缓冲比较寄存器；
- (6). 快速解码的中断向量；

7.2.2 与定时器 A 的比较（相同点与不同点）

- (1). TB 的计数长度可以选择（8、10、12、16BITS），而 TA 只有16位；

- (2). TB0CCRn 寄存器是双缓冲的，且可以分组；
- (3). 所有的 TB 输出可以被设为高阻状态；
- (4). TB 没有 SCCI，即捕获器输入信号 CCI 没有被锁存；

看结构图，观察和 TA 有什么区别：



7.2 Timer0_B 寄存器介绍及设置)

声明：所有寄存器同样支持字和字节操作

所有寄存器初始化都为0x0000

7.2.1 TB 控制寄存器 **TBOCTL**（最常用最基本）（和 TA 有一点不同）

rw-(0) 表示默认读写均为0

15	14	13	12	11	10	9	8
Unused	TBCLGRP _x			CNTL	Unused	TBSSEL	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ID		MC		Unused	TBCLR	TBIE	TBIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

TBCLGRP: TBOCL_n 分组控制

00: 每个 TBOCL_n 独立使用

01: TBOCL₁+TBOCL₂作为一组（TBOCCR₁的 CLLD 位控制整组数据更新）

TBOCL₃+TBOCL₄作为一组（TBOCCR₃的 CLLD 位控制整组数据更新）

TBOCL₅+TBOCL₆作为一组（TBOCCR₅的 CLLD 位控制整组数据更新）

10: TBOCL₁、2、3一组，（TBOCCR₁的 CLLD 位控制整组数据更新）

TBOCL₄、5、6一组，（TBOCCR₄的 CLLD 位控制整组数据更新）

11: TBOCL₀、1、2、3、4、5、6整合为一组，

（TBOCCR₁的 CLLD 位控制整组数据更新）

CNTL: 计数器长度控制

00 16位，即最大可以计到0FFFFh

01 12位，即最大可以计到0FFFh

10 10位，即最大可以计到03FFh

11 8位，即最大可以计到0FFh

TBSSEL : 时钟源选择。尽量不要选 TACLK 外部时钟源，因为如果 TACLK 和 CPU 时钟不同步，很容易出问题。（TBOCLK 接 P7.7 引脚）

00 TBCLK

01 ACLK

10 SMCLK

11 ~TBCLK

ID: 第一次分频控制。ID₀-1分频；ID₁-2分频；ID₂-4分频；ID₃-8分频

MC: 工作模式控制。（建议在修改定时器运行模式前先停止定时器（中断使能、中断标志、TACLR 例外），以避免产生未知的误操作。）（和 TA 一样）

00 停止模式：定时器停止

- 01 增模式： 定时器计数到 TBOCCRO
- 10 连续模式，定时器计数到 0FFFH（16位）...12位、10位...
- 11 增减模式：定时器加计数到 TBOCCRO 然后减计数到 0000H

TBCLR: 定时器清零位。该位置位会复位 TA 寄存器，时钟分频和计数方向。
TACLR 位会自动复位并置0

TBIE: 定时器中断使能
0: 中断禁止

1: 中断允许

TBIFG: 中断标志位
0: 没有中断发生

1: 有中断挂起

7.2.2 计数值存放寄存器 TBOR

7.2.3 扩展寄存器 TBEX0

很简单，这个寄存器就是为了控制时钟源的二次分频（看结构图）。
该寄存器的低3为定义为 TBIDEX：000-111 分别表示1-8分频

7.2.4 捕获/比较寄存器 TBCCR0-TBCCR6（共7个）

比较模式下，用来设定计数终值；

捕获模式下用来将捕获的 TBR 值存放在 TBCCR_x 中。

7.2.5 捕获/比较控制寄存器 TBOCCTL0-TBOCCTL6:

15	14	13	12	11	10	9	8
CM		CCIS		SCS	CLLD		CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
OUTMOD		CCIE		CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

CM: 捕获模式设定00 不捕获

01 上升沿捕获

10 下降沿捕获

11 上升和下降沿都捕获

CCIS: 捕获源的选择00 CCI_xA

01 CCI_xB

10 GND

11 VCC

SCS: 同步捕获源，设定是否与时钟同步

0 异步捕获

1 同步捕获

CLLD: 比较寄存器缓冲装载模式选择。

- 00 TBOCCRn 的值（改变时）立即装载到 TBOCLn
- 01 当 TBOR 的值计到0时，进行装载
- 10 增模式或者连续模式下，TBRO值计到0时，进行装载；

增减模式下，TBRO计数到 TBCL0时，开始装载；

- 11 TBRO计数到 TBCL0时，开始装载；

CAP: 0-比较模式 1-捕获模式

OUTMOD: 输出模式控制位。同 TA 一模一样

CCIE: 中断使能，该位允许相应的 CCIFG 标志中断请求。

0-中断禁止 1 -中断允许

CCI : 捕获比较输入，所选择的输入信号可以通过该位读取

OUT : 对于输出模式0，该位直接控制输出状态。

0-输出低电平 1-输出高电平

COV: 捕获溢出位。该位表示一个捕获溢出发出，COV 必须由软件复位。

0-没有捕获溢出发生1-有捕获溢出发生

CCIFG: 捕获比较中断标志位。

0-没有中断挂起1-有中断挂起

7.2.6 中断向量寄存器 TBOIV

同 TAIV 一样，里面存放一个数字编号。

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
0	0	0	0	TBIV			0
r0	r0	r0	r0	r-(0)	r-(0)	r-(0)	r0

TBIV

Bits 15-0

Timer_B interrupt vector value

TBIV Contents	Interrupt Source	Interrupt Flag	Interrupt Priority
00h	No interrupt pending		
02h	Capture/compare 1	TBxCCR1 CCIFG	Highest
04h	Capture/compare 2	TBxCCR2 CCIFG	
06h	Capture/compare 3	TBxCCR3 CCIFG	
08h	Capture/compare 4	TBxCCR4 CCIFG	
0Ah	Capture/compare 5	TBxCCR5 CCIFG	
0Ch	Capture/compare 6	TBxCCR6 CCIFG	
0Eh	Timer overflow	TBxCTL TBIFG	Lowest



7.3 重点讲 TB 和 TA 的不同之处

7.3.1 没有再把 CCI 信号锁存了

TA 作捕获器的时候，CCI 为捕获信号，然后 CCI 被锁存输出 为 SCCI；

但是，TB 没有锁存。也就是说只能从 CCI 位查看输入信号了。

7.3.2 计数值位数可调了（其实无所谓，都可以16位那干嘛不用）

TA 的计数值寄存器 TAR 只能是16位 (0xFFFFh);

TB 的计数值寄存器 TBR 可以选择是16、12、10、8位;

7.3.3 两级缓冲比较器（比较模式下）

TA 里面，我们在 TACCRn 中写入要比较的数值，然后让 TAR 中的计数值和 TACCRn 比较，如果相等了，相应的标志位就会置位;

TB 里面，不仅有 TBOCCRn，还多了一个二级缓存器 TBOCLn。TBOCLn 不能被直接进行操作，它的值只能来源于 TBOCCRn。计数的时候，TBOR 中的计数值不和 TBOCCRn 比较，而是和 TBOCLn 进行比较。

二级缓冲是为了防止我们在修改 TBOCCRn 的值的时候，对计数产生影响。因为计数器不直接和 TBOCCRn 比较，而是 TBOCCRn 把值赋给 TBOCLn，由 TBOCLn 去和 TBOR 进行比较。所以也就有了 CLLD 位控制比较寄存器缓冲装载模式：（当向 TBOCCRn 中重新写数时）

00 TBOCCRn 的值立即装载到 TBOCLn

01 当 TBOR 的值计到0时，进行装载

10 增模式或者连续模式下，TBRO值计到0时，进行装载；
增减模式下，TBRO计数到 TBCL0时，开始装载；

11 TBRO计数到 TBCL0时，开始装载；

7.3.4比较器可以被分组

TA 没有二级缓冲寄存器，而且本来的 TACCRn 也只能被单独使用。

对于 TB:

TBCLGGRP: TBOCLn 二级缓冲寄存器分组控制

00: 每个 TBOCLn 独立使用

01: TBOCL1+TBOCL2作为一组（**TBOCCR1**的 CLLD 位控制整组数据更新）

TBOCL3+TBOCL4作为一组（**TBOCCR3**的 CLLD 位控制整组数据更新）

TBOCL5+TBOCL6作为一组（**TBOCCR5**的 CLLD 位控制整组数据更新）

10: TBOCL1、2、3一组，（**TBOCCR1**的 CLLD 位控制整组数据更新）

TBOCL4、5、6一组，（**TBOCCR4**的 CLLD 位控制整组数据更新）

11: TBOCL0、1、2、3、4、5、6整合为一组，

（**TBOCCR1**的 CLLD 位控制整组数据更新）

所谓的分组，就是该组的数据要同时更新。

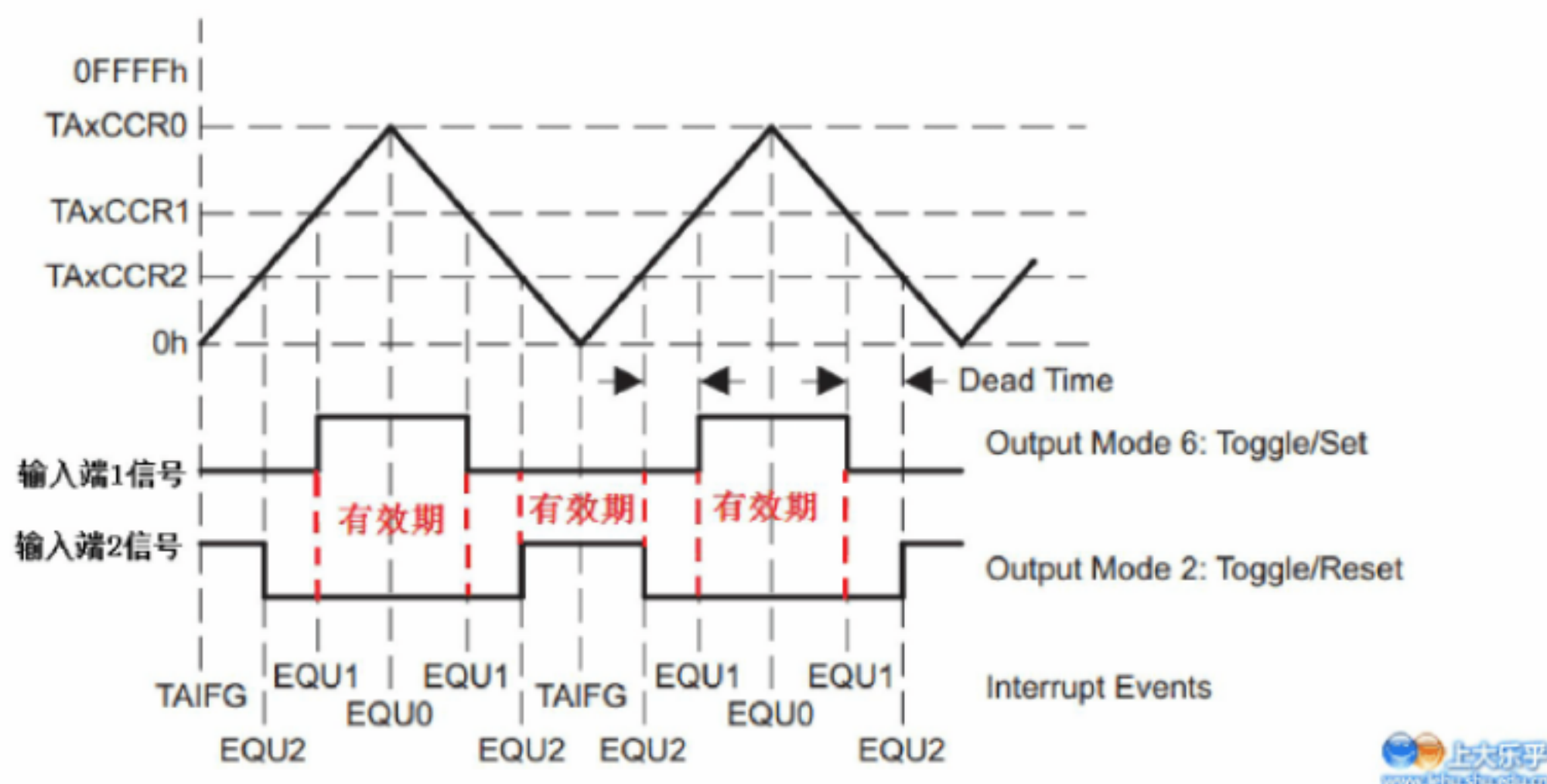
以10模式下的 TBOCL1、2、3这组为例：

TBOCCTL1中的 CLLD 设置为01，即 TBOR 计数到0时，TBOCCR1就会把值装载（更新）到 TBOCL2中，同时 TBOCCR2就会把值装载（更新）到 TBOCL2中，同时 TBOCCR3也会把值装载（更新）到 TBOCL3中。无论 TBOCCRn 中的值有几个发生

了变化，但它们都只会同时更新 TBOCLn。

一个重要的应用：

如图，增减模式下，可以用来产生有死区时间（dead time）的信号。



我们知道有一种 **H 桥** 电路结构，该电路有一般有两个输入端，而且这两个输入端（1和2）严禁同时输入高电平。那么该怎么产生驱动该电路的信号呢：但是，如果我们想要在系统运行的时候，修改死区时间怎么办。那只有修改比较器里面的数值了。这样就有问题了，如果两个比较器数值没有同时修改，那么就有可能产生同时为高电平的情况，这就很危险了。所以，我们把比较器分组，这样数据同时更新，就不会有这样的担心了。

MSP430F5529 番外（二）纠正 XT1配置错误

错误更正说明：

在第三章讲 UCS 时钟系统的时候，实验二是将 MCLK 和 ACLK 配置为 XT1（F5529 的 XT1为32.768KHZ）

当时的程序如下：

```
/*1. 配置 IO 口5.4和5.5为 XT1功能。*/  
  
/*2. 配置 XCAP 为 XCAP_3，即12PF 的电容。*/
```

```

/*3. 清除 XT10FF 标志位。*/

/*4. 等待 XT1起振。*/

#include<msp430.h>

void main(void) {

P1SEL |= BIT0;

P1DIR |= BIT0; //测量 ACLK 用

P2SEL |= BIT2;

P2DIR |= BIT2; //测量 SMCLK 用

P7SEL |= BIT7;

P7DIR |= BIT7; //测量 MCLK 用

P5SEL |= BIT4|BIT5; //配置为 XT1功能, 电路板上晶振接于这两脚

UCSCTL6 |= XCAP_3; //配置电容为12pF

UCSCTL6 &= ~XT10FF; //使能 XT1

/*下面是很重要的一步：*/

/* XT1刚刚起振的时候可能有错误, 导致时钟错误标志位置位, 必须先清零*/

/*OFIFG 即 Osc Fault Flag, 位于 SFRIFG1中*/

while(SFRIFG1 & OFIFG) //如果有时钟错误{

UCSCTL7 &= ~(XT2OFFG+DCOFFG+XT1LFOFFG); //清除3种时钟错误标志

SFRIFG1&= ~(OFIFG); //清除时钟错误标志位}

    UCSCTL4&=(UCSCTL4&(~(SELA_7|SELM_7)))|SELA_0|SELM_0;

    //将 SMCLK 和 MCLK 时钟源配置为 XT1

}

```

当时由于手里没有频率计，并没有测输出。

昨天实际测了一下，发现该程序存在问题：实际测量 MCLK 为876KHZ 左右，ACLK 虽然是

32.768KHZ, 但我感觉还是内部的 REF0。

解答如下: 这是官方的例程

```
ACLK = SMCLK = MCLK = XT1 = 32768

int main(void)
{
    WDCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    P1DIR |= BIT0; // ACLK set out to pins
    P1SEL |= BIT0;
    P2DIR |= BIT2; // SMCLK set out to pins
    P2SEL |= BIT2;
    P7DIR |= BIT7; // MCLK set out to pins
    P7SEL |= BIT7;

    P5SEL |= BIT4+BIT5; // Select XT1

    UCSCTL6 &= ~(XT1OFF); // XT1 On
    UCSCTL6 |= XCAP_3; // Internal load cap
    UCSCTL3 = 0; // FLL Reference Clock = XT1

    // Loop until XT1, XT2 & DCO stabilizes - In this case loop until XT1 and DCO settle
    do
    {
        UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG);
        // Clear XT2, XT1, DCO fault flags
        SFRIFG1 &= ~OFIFG; // Clear fault flags
    } while (SFRIFG1 & OFIFG); // Test oscillator fault flag

    UCSCTL6 &= ~(XT1DRIVE_3); // Xtal is now stable, reduce drive strength

    UCSCTL4 = SELA_0 + SELS_0 + SELM_0; // SMCLK = MCLK = ACLK = LFX1

}
```

注意划红线的语句, 要用等号赋值。因为默认 SELS 和 SELM 的值是不为0的, 所以用或赋值的话会出错。

默 认 值 请 看 下 表 :

Table 5-7. UCSCTL4 Register Description

Bit	Field	Type	Reset	Description
15-11	Reserved	R	0h	Reserved. Always reads as 0.
10-8	SELA	RW	0h	Selects the ACLK source 000b = XT1CLK 001b = VLOCLK 010b = REFOCLK 011b = DCOCLK 100b = DCOCLKDIV 101b = XT2CLK when available, otherwise DCOCLKDIV 110b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV. 111b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV.
7	Reserved	R	0h	Reserved. Always reads as 0.
6-4	SELS	RW	4h	Selects the SMCLK source 000b = XT1CLK 001b = VLOCLK 010b = REFOCLK 011b = DCOCLK 100b = DCOCLKDIV 101b = XT2CLK when available, otherwise DCOCLKDIV 110b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV. 111b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV.
3	Reserved	R	0h	Reserved. Always reads as 0.
2-0	SELM	RW	4h	Selects the MCLK source 000b = XT1CLK 001b = VLOCLK 010b = REFOCLK 011b = DCOCLK 100b = DCOCLKDIV 101b = XT2CLK when available, otherwise DCOCLKDIV 110b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV. 111b = Reserved for future use. Defaults to XT2CLK when available, otherwise DCOCLKDIV.

八、实时时钟 RTC_A

实时时钟模块提供了具有日历模式的时钟计数、灵活可编程的闹钟以及可校准的时钟计数器。

8.1 RTC_A 简介

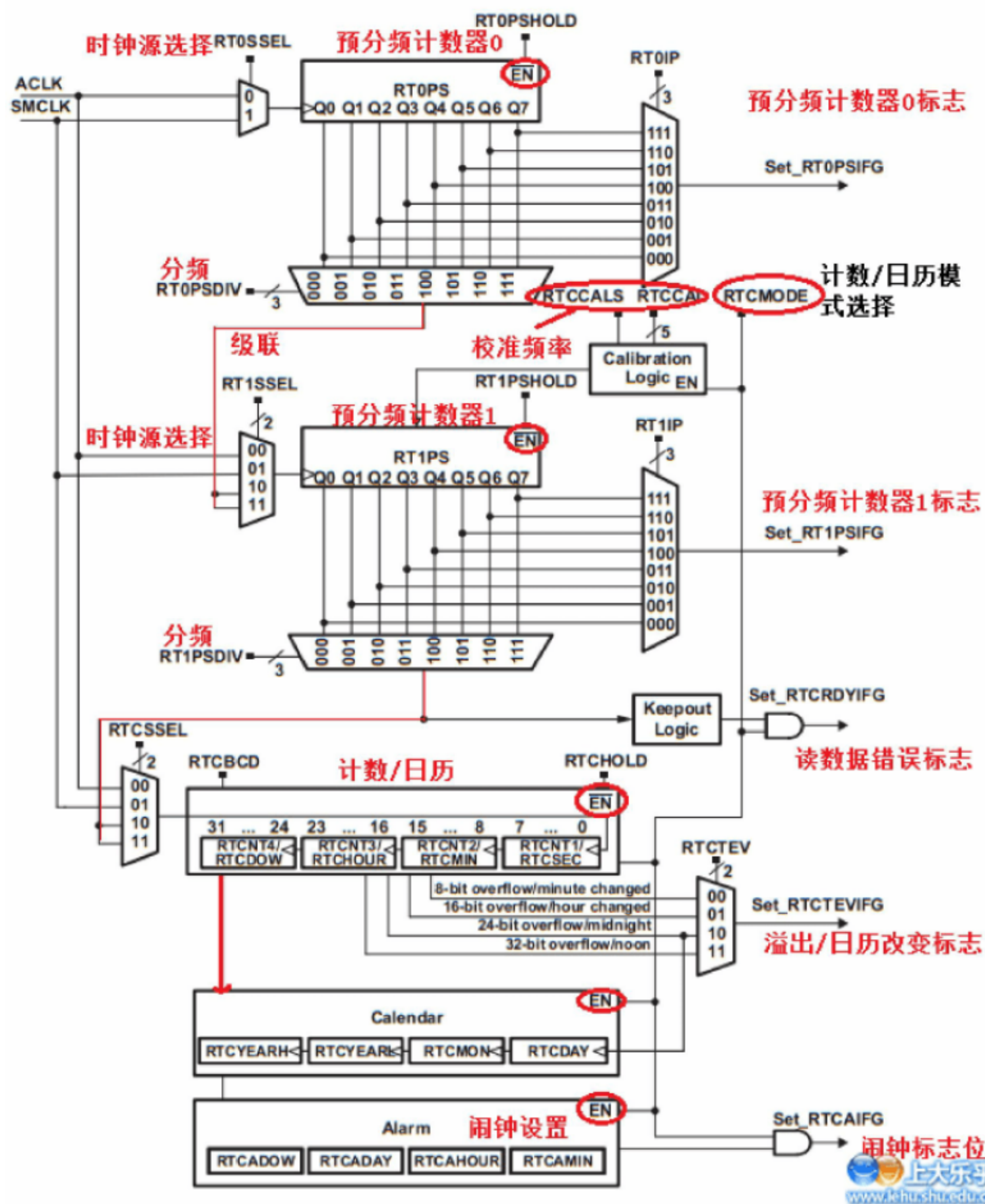
实时时钟模块提供了一个可以配置成一般目的计数器的日历时钟。

RTC_A 的特点包括：

- (1). 可配置成带日历的实时时钟或者一般目的的计数器；
- (2). 在日历模式中提供了秒钟，分钟，小时，星期，日期，月份和年份；
- (3). 具有中断能力；
- (4). 实时时钟模式下可以配置为 BCD 或者二进制模式；
- (5). 实时时钟模式下具有可编程的闹钟；
- (6). 实时时钟模式里具有时间偏差的逻辑校正；

注意：实时时钟模块的大多数寄存器没有初始条件。
在使用这个模块之前，用户必须通过软件对寄

寄存器进行配置。



8.2 RTC_A 的寄存器

说实话，RTC 这一章不太好写，东西太多又太碎，不好总结。它一个人的寄存器，比前面加起来似乎还要多。不过还好控制寄存器只有几个，大部分为数值寄存器。

(大家浏览下列寄存器时，注意和上面的结构图配合)

贴这张图的目的**仅仅**在于告诉大家，16位的寄存器大部分被分成两个8位的寄存器了，操作的时候需要注意你的操作对象是8位的还是16位的。**在这里常用8位寄存器来操作。(以下如未特意声明，则全部为8位寄存器)**

Register	Short Form	Register Type	Register Access	Address Offset	Initial State
Real-Time Clock Control 0, 1	<u>RTCCTL01</u>	Read/write	Word		4000h
Real-Time Clock Control 0	<u>RTCCTL0 or RTCCTL01_L</u>	Read/write	Byte		00h
Real-Time Clock Control 1	<u>RTCCTL1 or RTCCTL01_H</u>	Read/write	Byte		40h
Real-Time Clock Control 2, 3	<u>RTCCTL23</u>	Read/write	Word		0000h
Real-Time Clock Control 2	<u>RTCCTL2 or RTCCTL23_L</u>	Read/write	Byte		00h
Real-Time Clock Control 3	<u>RTCCTL3 or RTCCTL23_H</u>	Read/write	Byte		00h
Real-Time Prescale Timer 0 Control	<u>RTCPS0CTL</u>	Read/write	Word		0100h
	<u>RTCPS0CTL_L or RTCPS0CTL_L</u>	Read/write	Byte		00h
	<u>RTCPS0CTL_H or RTCPS0CTL_H</u>	Read/write	Byte		01h
Real-Time Prescale Timer 1 Control	<u>RTCPS1CTL</u>	Read/write	Word		0100h
	<u>RTCPS1CTL_L or RTCPS1CTL_L</u>	Read/write	Byte		00h
	<u>RTCPS1CTL_H or RTCPS1CTL_H</u>	Read/write	Byte		01h
Real-Time Prescale Timer 0, 1 Counter	<u>RTCPS</u>	Read/write	Word		undefined
Real-Time Prescale Timer 0 Counter	<u>RT0PS or RTCPS_L</u>	Read/write	Byte		undefined
Real-Time Prescale Timer 1 Counter	<u>RT1PS or RTCPS_H</u>	Read/write	Byte		undefined
Real Time Clock Interrupt Vector	<u>RTCIV</u>	Read	Word		0000h
	<u>RTCIV_L</u>	Read	Byte		00h
	<u>RTCIV_H</u>	Read	Byte		00h
Real-Time Clock Seconds, Minutes/ Real-Time Counter 1, 2	<u>RTCTIM0 or RTCNT12</u>	Read/write	Word		undefined
Real-Time Clock Seconds/ Real-Time Counter 1	<u>RTCSEC /RTCNT1 or RTCTIM0_L</u>	Read/write	Byte		undefined
Real-Time Clock Minutes/ Real-Time Counter 2	<u>RTCMIN/RTCNT2 or RTCTIM0_H</u>	Read/write	Byte		undefined
Real-Time Clock Hour, Day of Week/ Real-Time Counter 3, 4	<u>RTCTIM1 or RTCNT34</u>	Read/write	Word		undefined
Real-Time Clock Hour/ Real-Time Counter 3	<u>RTCHOUR/RTCNT3 or RTCTIM1_L</u>	Read/write	Byte		undefined
Real-Time Clock Day of Week/ Real-Time Counter 4	<u>RTCDOWRTCNT4 or RTCTIM1_H</u>	Read/write	Byte		undefined

Real-Time Clock Date	RTCDATE	Read/write	Word	undefined
Real-Time Clock Day of Month	RTCDAY or RTCDATE_L	Read/write	Byte	undefined
Real-Time Clock Month	RTCMON or RTCDATE_H	Read/write	Byte	undefined
Real-Time Clock Year	RTCYEAR	Read/write	Word	undefined
	RTCYEARL or RTCYEAR_L	Read/write	Byte	undefined
	RTCYEARH or RTCYEAR_H	Read/write	Byte	undefined
Real-Time Clock Minutes, Hour Alarm	RTCAMINHR	Read/write	Word	undefined
Real-Time Clock Minutes Alarm	RTCAMIN or RTCAMINHR_L	Read/write	Byte	undefined
Real-Time Clock Hours Alarm	RTCAHOUR or RTCAMINHR_H	Read/write	Byte	undefined
Real-Time Clock Day of Week, Day of Month Alarm	RTCADOWDAY	Read/write	Word	undefined
Real-Time Clock Day of Week Alarm	RTCADOW or RTCADOWDAY_L	Read/write	Byte	undefined
Real-Time Clock Day of Month Alarm	RTCADAY or RTCADOWDAY_H	Read/write	Byte	undefined



RTCCTL0 实时时钟控制寄存器0 (r0表示读为0)

7	6	5	4	3	2	1	0
Reserved	RTCTEIVE	RTCAIE	RTCRDYIE	Reserved	RTCTEVIFG	RTCAIFG	RTCRDYIFG
r0	rw-0	rw-0	rw-0	r0	rw-(0)	rw-(0)	



RTCTEIVE: 实时时钟-时间事件中断使能

0: 禁止中断

1: 允许中断

RTCAIE: 实时时钟-闹钟中断使能, 在计数器模式时被清除 (RTCMODE = 0)

0: 禁止中断

1: 允许中断

RTCRDYIE: 实时时钟读取准备中断使能

0: 禁止中断

1: 允许中断

RTCTEVIFG: 实时时钟-时间事件标志

0: 没有时间事件发生

1: 有时间事件发生

RTCAIFG: 实时时钟-闹钟标志位, 在计数器模式时被清除 (RTCMODE = 0)

0: 没有时间事件发生

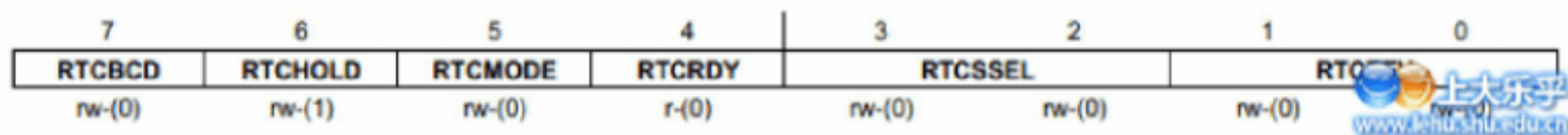
1: 时间事件发生

RTCRDYIFG: 实时时钟读取准备标志位

0: 实时时钟不能被安全读取

1: 实时时钟能被安全读取

RTCCTL1 实时时钟控制寄存器1 (rw-(0)表示读写为0)



RTCBCD: 实时时钟 BCD 码格式选择位，选择实时时钟的 BCD 格式。只能应用于日历模式（RTCMODE = 1），在计数器模式中会被忽略设置。改变这个位会将秒、分、小时、星期和年清零，将日期和月份置1。之后，实时时钟寄存器必须被软件设置。

0：选择2进制或者十六进制

1：选择 BCD 码

RTCHOLD: 实时时钟保持位

0：实时时钟（32位计数器或者是日历模式）正在运作

1：计数器模式（RTCMODE = 0），该位置1只会使32位计数器停止；在日历模式（RTCMODE = 1）日历以及预分频计数器会被停止。RTOPS 和 RT1PS、RTOPSHOLD 和 RT1PSHOLD 位可以忽略。

RTCMODE: RTC 模式选择。

0： 32位计数器模式

1： 日历模式。在日历模式和计数器模式之间的切换会重置实时时钟；计数器模式切换到日历模式会将秒、分、小时、星期和年清零，将日期和月份置1。实时时钟寄存器需要后来被软件设置。RTOPS 和 RT1PS 也会被清零。

RTCRDY: 实时时钟准备位

0：实时时钟值在转换过渡（日历模式）

1： 实时时钟值可被安全读取（日历模式）。在计数器模式，RTCRDY 保持清除。

RTCSEL: 实时时钟源选择位。选择时钟源输入到 RTC/32计数器。在 RTC 日历模式这两位是不考虑的，其输入默认是 RT1PS 的输出。

00 ACLK

01 SMLK

10从 RT1PS 输出

11 从 RT1PS 输出

RTCTEV: RTC 时间事件指示

RTC 模式	RTCTEV 的值	内部中断
计数器模式	00	8位溢出
	01	16位溢出
	10	24位溢出
	11	32位溢出
日历模式	00	分钟改变
	01	小时改变
	10	午夜（00:00）
	11	白天（12:00）

RTCCTL2 实时时钟控制寄存器2 (rw-(0)表示读写为0)

7	6	5	4	3	2	1	0
RTCCALS	Reserved	RTCCAL					
rw-(0)	r0	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

RTCCALS: RTC 校准标志

0 : 频率调整下降

1 : 频率调整上升

RTCCAL: RTC 频率校准位

每个 LSB 位代表大约 +4PPM(RTCCALS=1) 或 -2PPM(RTCCALS=0) 调整频率。

RTCCTL3 实时时钟控制寄存器3

7	6	5	4	3	2	1	0
Reserved						RTCCALF	
r0	r0	r0	r0	r0	r0	rw-(0)	rw-(0)

RTCCALF: RTC 校准频率 校准测量时选择频率输出到 RTCCLK 引脚(P2.6)上。相对应的端口必须配置为外围模块功能(即 P2SEL=BIT6)。RTCCLK 在计数模式不可用,此时保持为低且 RTCCALF 位的值不确定。

00: 没有频率输出到 RTCCLK 引脚

01: 512HZ

10: 256HZ

11: 1HZ

RTCPSoCTL 预分频定时器0控制寄存器(16位寄存器)

15	14	13	12	11	10	9	8
Reserved	RTOSSEL	RTOPSDIV			Reserved	Reserved	RTOPSHOLD
r0	rw-0	rw-0	rw-0	rw-0	r0	r0	rw-1
7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	RT0IP			RT0PSIE	RT0PSIFG
r0	r0	r0	rw-0	rw-0	rw-0	rw-0	rw-0

RTOSSEL: 预分频定时器0时钟源选择位。选择时钟源输入到 RTOPS 计数器。在 RTC 日历模式该位不用关心,因为 RTOPS 时钟输入自动设置到 ACLK (32.768KHZ)。

0: ACLK

1: SMCLK

RTOPSDIV: 预分频定时器0分频,这些位控制 RTOPS 计数器的分频。在 RTC 日历模式对于 RTOPS 和 RT1PS 这些位是不用关心的,因为 RTOPS 时钟输出自动设置到256分频。RT1PS 时钟输出自动设置到128分频。

000-111, 分别表示2、4、8、16、32、64、128、256分频

RTOPSHOLD: 预分频定时器0保持位。在 RTC 日历模式这些位是不关心的。RTCHOLD 置位时,RTOPS 停止工作。

RT0IP: 预分频器0中断间隔。

000-111, 分别表示2、4、8、16、32、64、128、256分频

RT0PSIE: 预分频定时器0中断允许

0: 中断不允许

1: 中断允许

RT0PSIFG: 预分频定时器0中断标志

0: 没有定时事件发生

1: 有定时事件发生

RTCPS1CTL 预分频定时器1控制寄存器(16位寄存器)

15	14	13	12	11	10	9	8	
RT1SSEL		RT1PSDIV				Reserved	Reserved	RT1PSHOLD
rw-0		rw-0		rw-0		r0		rw-1
7	6	5	4	3	2	1	0	
Reserved	Reserved	Reserved	RT1IP				RT1PSIE	RT1PSIFG
r0		r0		r0		rw-0		rw-0

RT1SSEL: 预分频定时器1时钟源选择位。选择时钟源输入到 RT1PS 计数器。在 RTC 日历模式该位不用关心, 因为 RT1PS 时钟输入自动设置为 RT0PS 的输出。

00: ACLK

01: SMCLK

10: 从 RT0PS 输出

11: 从 RT0PS 输出

(其余项同 RTCPS0CTL)

上面的都为控制寄存器, 操作比较复杂, 下面讲到的都是数值寄存器。

RTCNT1: RTC 计数寄存器1, 计数器模式

RTCNT2: RTC 计数寄存器2, 计数器模式

RTCNT3: RTC 计数寄存器3, 计数器模式

RTCNT4: RTC 计数寄存器4, 计数器模式

RTCSEC: 秒寄存器。BCD/BIN

先说明一下 BCD/BIN 的意思(下面同理)

前面提到可以控制日历模式的数值寄存器中的数值用 BCD 码或者二进制码表示。以 RTCSEC 为例:

当选作二进制模式时:(低6位就可以表示秒1-60)

7	6	5	4	3	2	1	0
0	0	Seconds (0 to 59)					
r-0	r-0	rw	rw	rw	rw	rw	

当选作 BCD 模式时:(低4位表示秒的各位0-9; 4-6位表示秒的

十位0-6，高位不用时默认为0)



RTCMIN: 分寄存器, BCD/BIN

RTCHOUR: 时寄存器, BCD/BIN

RTCDOW: 星期日数寄存器, 因为只有1-7, 所以无所谓 BCD/BIN 了

RTCDAY: 日寄存器, BCD/BIN

RTCMON: 月寄存器, BCD/BIN

RTCYEARL: 年低字节寄存器, 个年位以及十年位, BCD/BIN

RTCYEARH: 年高字节寄存器, 百年位以及千年位, BCD/BIN

RTCAMIN: 分闹铃寄存器, BCD/BIN

RTCAHOUR: 时闹铃寄存器, BCD/BIN

RTCADOW: 星期闹铃寄存器

RTCADAY: 日闹铃寄存器, BCD/BIN

//上述4个闹钟寄存器的最高位都为使能位 AE, 置位时相应寄存器才有效

RTOPS: 预分频定时器0计数值

RT1PS: 预分频定时器1计数值

RTCIV: 中断向量值寄存器 (16位寄存器)

RTCIV 内容	中断源	中断标志	中断优先级
00H	没中断发生		
02H	RTC 读	RTCRDYIFG	高
04H	RTC 间隔定时器	RTCDEVIFG	
06H	RTC 用户闹钟	RTCAIFG	
08H	RTC 预分频 0	RT0PSIFG	
0AH	RTC 预分频 1	RT1PSIFG	
0CH	保留		
0EH	保留		
10H	保留		低

8.3 RTC 的各种操作流程

8.3.1 计时器模式

(1). RTCMODE 位置0, 进入32位计时器模式;

从日历模式切换到计数器模式会将计数值寄存器 (RCTNT1, RCTNT2, RCTNT3, RCTNT4) 和预换算计数器 (RTOPS, RT1PS) 全部清零

(2). 选择时钟源, 并设计二级分频;

计数器的时钟可源于 ACLK、SMCLK 或者是 RT1PS 的输出。当使用 RT1PS 的输出作为计数源的时候，一定要先将 RTOPS，RT1PS 的 HOLD 位清零，使其可以正常工作，然后再分别配置二者的分频数。

再者，两个预分频器 RTOPS 和 RT1PS 也可以作为独立的计数器来用（级联成16位也是可以的）。通过 RT0IP 和 RT1IP 可以设置间隔。比如，选择 ACLK 32768HZ 作为时钟，间隔设置为256，也就是说每当计数器计到 $32768/256=128$ 的整数倍时，该标志位就会置位。

(3). 32位计数器是由4个8位计数器级联而成，这能提供8位、16位、24位、32位溢出间隔。RTCDEV 位选择触发哪一个溢出间隔，通过设置 RTCDEVIE 位，一个 RTCDEV 发生能够触发一个中断。计数器 RTCNT1到 RTCNT4，每一个都可以单独的访问，并可能被写入。

(4). 如何关闭计数器。

为了简单一点，把所有 HOLD 位都置位，则可以保证在任何情况下都可以关闭32位计数器。

注意：对计数值寄存器写时，立即生效。

读时，如果该时钟与 CPU 时钟不同步，则需要暂停计数器来读数。或者通过多次读取，来软件判断哪个是正确值。

8.3.2 日历模式

当 RTCMODE 置位的时候，日历模式就被选中了。在日历模式中，实时时钟模块可选择以 BCD 码或者是十六进制提供秒、分、小时、星期、日期、月份和年份。日历会自动计算是否是闰年，这个算法可以精确到1901年到2099年。

(1). 时钟和预分频。

RTOPS 必须源于 ACLK，ACLK 必须是32768Hz，。RTOPS 会自动进行256分频，然后其输出再接 RT1PS，RT1PS 在被自动128分频，最后提供的时钟信号就是间隔一秒了。从计数器模式切换到日历模式时，会将秒、分、小时、星期、年份全部置清零，会将日期和月份全部置1。另外，RTOPS 和 RT1PS 也会被清零。（这里把这些状态暂定义为默认复位状态）

(2). 日历寄存器编码格式。

当 RTCBCD=1时，日历寄存器就会被选为 BCD 码格式。必须在时间设置之前选择好格式。改变 RTCBCD 的状态会使进入默认复位状态。

在日历模式下，RTOSSEL、RT1SSEL、RTOPSDIV、RT1PSDIV、RTOPSHOLD、RT1PSHOLD 和 RTCSEL 位都可以被忽略。置位 RTCHOLD 则会停止实时计数器、分频计数器和 RTOPS 、RT1PS。

(3). 灵活的闹钟

用户可编程闹钟功能只有在日历模式运行的时候才有效。

每一个闹钟寄存器都包括一个闹钟使能位，AE 可用来使能每一个闹钟寄存器。通过设置各式各样闹钟寄存器的 AE 位，可以生成多种闹钟。

比如说，一个用户需要在每一小时的15分钟（也就是00: 15: 00、01: 15: 00、02: 15: 00等等时刻）进行一次闹钟。这只要将 RTCAMIN 设置成15即可实现上述功能要求。通过置位 RTCAMIN 的 AE 位和清零闹钟寄存器的所有其它 AE 位，就会使能闹钟。正常工作时，对应的闹钟标志位 RTCCIFG 就会在00: 14: 59到00: 15: 00、01: 14: 59到01: 15: 00、02: 14: 59到02: 15: 00等等时刻被置位。

注意：写时间时，请务必保证格式正确，否则会出现无法预知的错误；

此外，修改闹钟时间的时候，为了避免错误，请先清 RTCAIE、RTCAIFG、AE 位来暂停闹钟功能。

8.3.3 读写日历模式下的 RTC 寄存器

因为系统时钟实际上是和实时时钟的时钟源是异步的，因此在进入实时时钟寄存器的时候要格外小心。

在日历模式下，实时时钟寄存器每秒钟更新一次。为了防止在更新的时候读取实时时钟数据而造成错误数据的读取，系统设立了一个禁止读取的区域。每次 RTC 寄存器更新的那一刹那，左右1/256s 被划为禁止读写的区域。RTCRDY 位用来指示这个时间区域。RTCRDY 置0时，表明处于这一区域；置1时表明在这一区域之外，可以发生读写。

一个简单而安全读取实时时钟寄存器的方法是利用 RTCRDYIFG 中断标志位。

设置 RTCRDYIE 使能 RTCRDYIFG 中断。一旦中断使能，在 RTCRDY 位上升沿的时候将会产生中断，致使 RTCRDYIFG 被置位。这样，我们几乎有一秒钟的安全时间去读写任一个寄存器。当中断得到响应的时候，RTCRDYIFG 会自动复位，当然也可以软件复位。

8.3.4 RTC 中断表

- (1). 每一个中断标志都配有相应的中断使能。
- (2). 请注意：RTCTE 定义的时间事件（计时模式和日历模式不同）
- (3). 对于 RTOPSIFG 和 RT1PSIFG 标志位，举一个例子：

通过 RT0IP 位，可以选择地让 RTOPSIFG 位用来生成间接中断。在日历模式下，RTOPS 的时钟源是32768Hz 的 ACLK，所以通过 RT0IP 控制中断间隔可以产生16384Hz、8192Hz、4096Hz、2048Hz、1024Hz、512Hz、256Hz 和128 Hz 的时间间隔。设置 RTOPSIE 位可以使能中断。

RTCIV 内容	中断源	中断标志	中断优先级
00H	没中断发生		
02H	RTC 读	RTCRDYIFG	高
04H	RTC 间隔定时器	RTCTEVIFG	
06H	RTC 用户闹钟	RTCAIFG	
08H	RTC 预分频 0	RT0PSIFG	
0AH	RTC 预分频 1	RT1PSIFG	
0CH	保留		
0EH	保留		
10H	保留		低

上大乐乎
www.shdledu.cn

8.3.5 RTC 校准

- (1). 把 P2.6 设定为输出状态，并声明有特殊功能；
P2.6 为 RTCCLK 的输出引脚
- (2). 通过设置 RTCCTL3 寄存器中的 RTCCALF 来设置 P2.6 输出信号的频率；
- (3). 精确测量该频率，然后计算误差；
- (4). 最后设置 RTCCTL2 寄存器，来调节频率增高或降低多少。

PPM 表示百万分之一所输出的频率；

注意：校准设置发生改变时，在 RTCCLK 引脚观察 512Hz 和 256Hz 的输出频率是不会有影响的。而校准发生改变时，1Hz 的输出频率是有影响的。

番外三：讲一些最近遇到的问题以及中断系统的说明

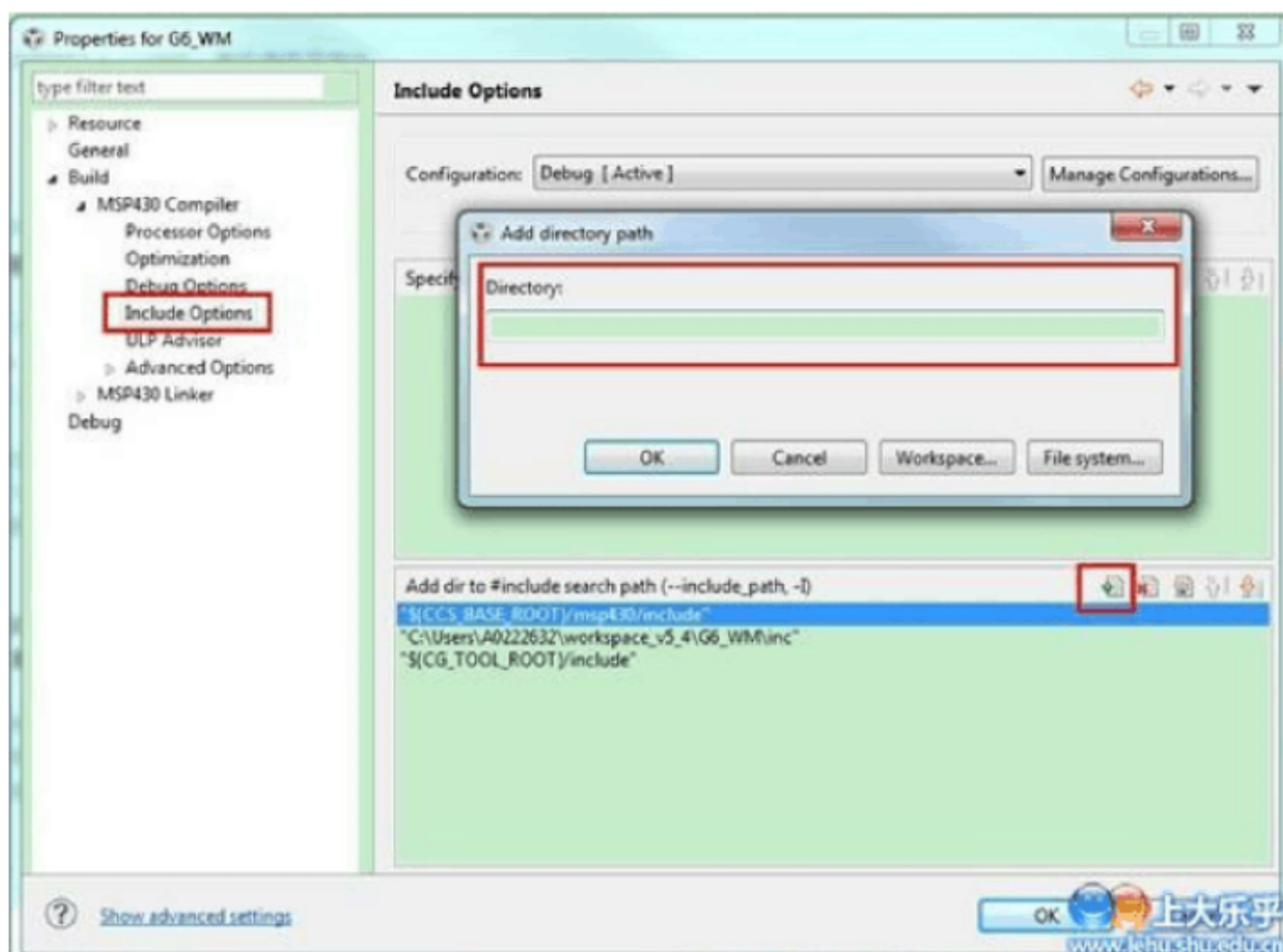
(1) CCSV5 中变量长度。

不同的芯片或者不同的编译环境下，变量长度的定义也是不同的。一般情况下，大家对变量长度也都不是很在意。但是，在做测量或者节约内存的时候，就有必要了，否则很可能造成变量溢出或者浪费空间。下表给出常用的几个变量类型的长度：默认值请看下表：

Type	Size	Representation	Range	
			Minimum	Maximum
char, signed char	8 bits	ASCII	-128	-127
unsigned char, bool	8 bits	ASCII	0	255
short, signed short	16 bits	2s complement	-32 768	32 767
unsigned short, wchar_t	16 bits	Binary	0	65 535
int, signed int	16 bits	2s complement	-32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	16 bits	2s complement	-32 768	32 767
float	32 bits	IEEE 32-bit	1.175 495e-38 ⁽¹⁾	3.40 282 35e+38
double	32 bits	IEEE 32-bit	1.175 495e-38 ⁽¹⁾	3.40 282 35e+38
long double	32 bits	IEEE 32-bit	1.175 495e-38 ⁽¹⁾	3.40 282 35e+38
pointers, references, pointer to data members	16 bits	Binary	0	0xFFFF
MSP430X large-data model pointers, references, pointer to data members ⁽²⁾	20 bits	Binary	0	0xFFFFF
MSP430 function pointers	16 bits	Binary	0	0xFFFF
MSP430X function pointers ⁽³⁾	20 bits	Binary	0	0xFFFFF



(2) 大家新建工程的时候，有时候会在工程里面一个个添加很多头文件以及相应的源文件。如液晶显示头文件 `HAL_Dogs102x6.h`，这个头文件很坑爹，想要用它，那么就不得不一个个添加十几个相关联的文件。于是，问题就来了，但你头文件添加太多的时候，编译很可能就会报错：“program will not fit into”，意思大概就是内存不够。内存一般来说不会不够用的。问题可能在于：添加头文件的时候没有设置路径：



或者最简便的方法就是：把官方的 LAB 例程中的主函数换成你的，其余不变，这样会很方便（不用一个个添加头文件了）。可能依然会出现上述问题，这时候只需要换一个不同编号（1-7）的 LAB 文件再“偷天换日”就可以了。

（3）MSP430F5529 不是自带 LCD Driver 的，是通过 SPI 来控制外部的点阵液晶显示的，并通过 P7.6 输出 PWM 来控制液晶背光的。

但是由于，P7.6 正好是 TB0.4，即定时器 TimerB0 的 4 通道，因此使用 TB0 时，极有可能会更改其 CCR0 的值。导致输出 PWM 波周期改变了。其结果就是导致液晶屏屏幕变暗。

（4）对于 430 而言，存在各种各样的中断。中断有时候确实很方便，但是对于中断使用，我觉得应该尽量少用尤其是中断嵌套的情况，更应该避免发生。因为多中断时，很可能出现一些难以控制和预料的意外。

当同时有多个中断来的时候才有优先级的考虑（优先级顺序可查看下面向量表）。

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power-Up External Reset Watchdog Timeout, Password Violation Flash Memory Password Violation	WDTIFG, KEYV (SYSRSTIV) ⁽¹⁾⁽²⁾	Reset	0FFFEh	63, highest
System NMI PMM Vacant Memory Access JTAG Mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRILIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV) ⁽¹⁾	(Non)maskable	0FFFCCh	62
User NMI NMI Oscillator Fault Flash Memory Access Violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV) ⁽¹⁾⁽²⁾	(Non)maskable	0FFFAh	61
Comp_B	Comparator B interrupt flags (CBIV) ⁽¹⁾⁽³⁾	Maskable	0FFF8h	60
TB0	TB0CCR0 CCIFG0 ⁽³⁾	Maskable	0FFF6h	59
TB0	TB0CCR1 CCIFG1 to TB0CCR6 CCIFG6, TB0IFG (TB0IV) ⁽¹⁾⁽³⁾	Maskable	0FFF4h	58
Watchdog Timer_A Interval Timer Mode	WDTIFG	Maskable	0FFF2h	57
USCI_A0 Receive or Transmit	UCA0RXIFG, UCA0TXIFG (UCA0IV) ⁽¹⁾⁽³⁾	Maskable	0FFF0h	56
USCI_B0 Receive or Transmit	UCB0RXIFG, UCB0TXIFG (UCB0IV) ⁽¹⁾⁽³⁾	Maskable	0FFEEh	55
ADC12_A	ADC12IFG0 to ADC12IFG15 (ADC12IV) ⁽¹⁾⁽³⁾⁽⁴⁾	Maskable	0FFECCh	54
TA0	TA0CCR0 CCIFG0 ⁽³⁾	Maskable	0FFEAh	53
TA0	TA0CCR1 CCIFG1 to TA0CCR4 CCIFG4, TA0IFG (TA0IV) ⁽¹⁾⁽³⁾	Maskable	0FFE8h	52
USB_UBM	USB interrupts (USBIV) ⁽¹⁾⁽³⁾	Maskable	0FFE6h	51
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV) ⁽¹⁾⁽³⁾	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0 ⁽³⁾	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV) ⁽¹⁾⁽³⁾	Maskable	0FFE0h	48
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDEh	47
USCI_A1 Receive or Transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDCh	46
USCI_B1 Receive or Transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV) ⁽¹⁾⁽³⁾	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0 ⁽³⁾	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD6h	43
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV) ⁽¹⁾⁽³⁾	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV) ⁽¹⁾⁽³⁾	Maskable	0FFD2h	41
Reserved	Reserved ⁽⁵⁾		0FFD0h	40
		
			0FF80h	...

有中断响应以后自动关闭总中断，这个时候即使来更高优先级的中断都不会响应。

要中断嵌套的话，就必须在中断中打开总中断。

实现中断嵌套需要注意以下几点：

1) 430默认的是关闭中断嵌套的，除非你在一个中断程序中再次开总中断 EINT；

2) 当进入中断程序时，只要不在中断中再次开中断，则总中断是关闭的，此时来中断不管是比当前中断的优先级高还是低都不执行；

3) 若在中断 A 中开了总中断，则可以响应后来的中断 B（不管 B 的优先级比 A 高还是低），B 执行完再继续执行 A。注意：进入中断 B 后总

中断同样也会关闭，如果 B 中断程序执行时需响应中断 C，则此时也要开总中断，若不需响应中断，则不用开中断，B 执行完后跳出中断程序进入 A 程序时，总中断会自动打开；

4) 若在中断中开了总中断，后来的中断同时有多个，则会按优先级来执行，即中断优先级只有在多个中断同时到来时才起作用！中断服务不执行抢先原则。

5) 对于单源中断，只要响应中断，系统硬件自动清中断标志位，对于 TA/TB 定时器的比较/捕获中断，只要访问 TAIV/TBIV，标志位便被自动清除；

(5) 如何将数字转化为对应的字符

大家以后经常会碰到如何将自己得到的一个整形或者浮点型数据显示到 LCD 上面去，LCD 只能识别 ASCII 码，不能直接识别数字。因此，就要用到将数字转化为 ASCII 码的。（一位一位转化）

a) 直接按照 ASCII 码的规则，把十进制数加上 0x30 就得到了相应的 ASCII 码；

b) 建一个字符数组 CharCode[10] = "0123456789"，
则 CharCode[i]，就是十进制数字 i 对应的 ASCII 码；

c) C 语言中 stdlib.h 头文件中定义的有将数值转化为字符串的函数，可以调用，不过个人不推荐；