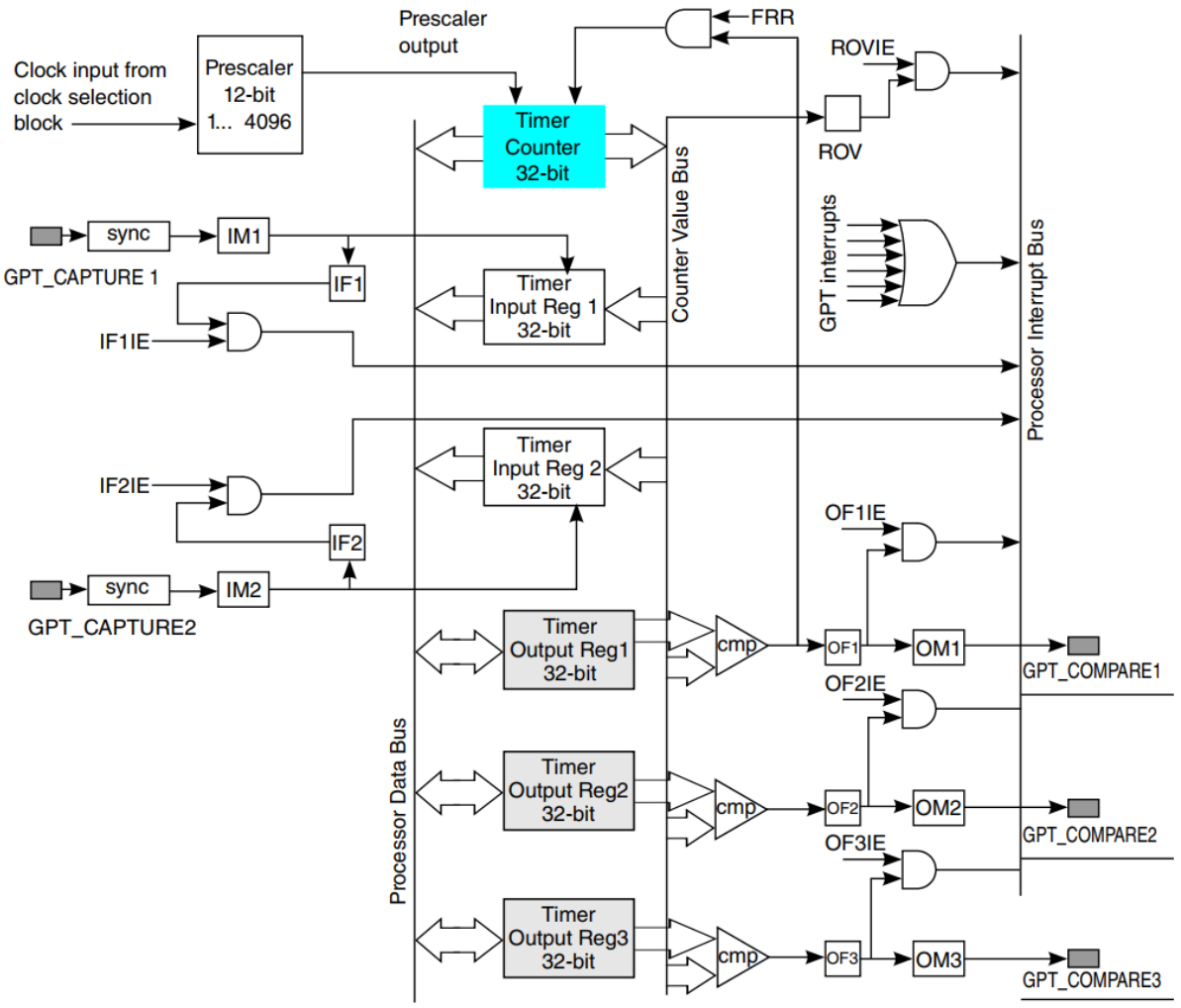
IMXRT定时器

(作者:TaterLi)

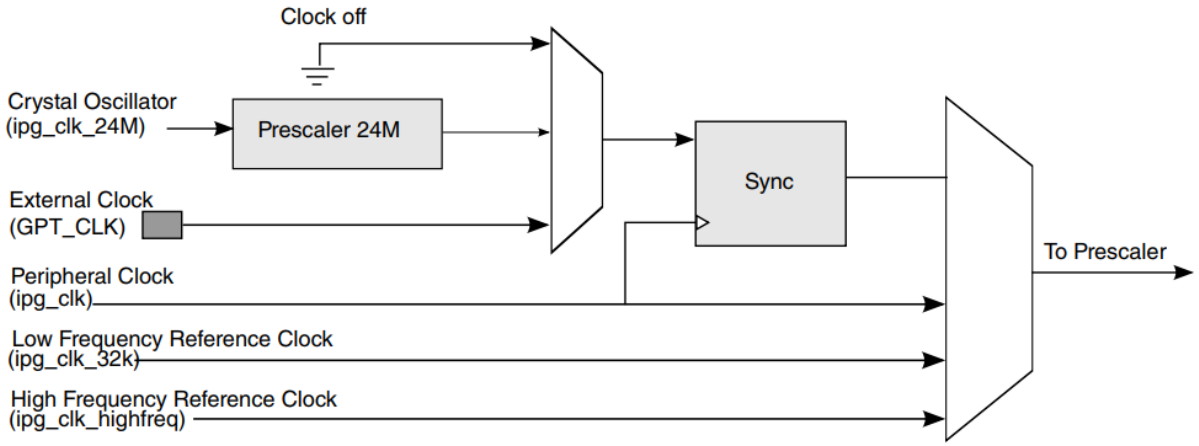
- 通用定时器(GPT),可以选择多种时钟源,具有12位预分频器的32位向上计数器,可以捕获引脚上的事件,捕获触发器可以设置为上升或者下降沿触发,也可以用输出比较输出一些简单的波形.
- 周期性中断计时器(PIT):基本的32位计数器,可编程分频,中断和从模式,可以链式连接起来,时钟源是外设时钟,可以用来当作系统的运
- 行时间计算器等等.
- 增强型PWM定时器(eFlexPWM):有捕获功能的PWM专用定时器,可以生成多路PWM(其实也就PWMA,PWMB,PWMX三个引脚),功能很多,也很复杂.
- 看门狗定时器(WDOG):如果没有喂狗,WDOG1向系统置位内部系统复位信号WDOG_RESET_B_DEB发送复位,WDOG2向SNVS发起中断(其实是为了TrustZone,但是这个芯片没有),WDOG3是独立看门狗,可以使用不同的时钟源,比如1MHz RC OSC,没喂狗的时候也是复位MCU,当然,这些WDOG3还是窗口看门狗.
- 外部看门狗监视器(EWM):用于监视外部电路,如果EWM复位,就会在某个引脚输出脉冲,可以选择多种时钟源.
- RTC:😂 (表示定时器家族没有这个玩意)

General Purpose Timer (GPT)

比较复杂,功能比较多,有输入(2个)输出(3个)功能,可以选择时钟输入,有分频,可以产生比较输出,可以产生捕获.

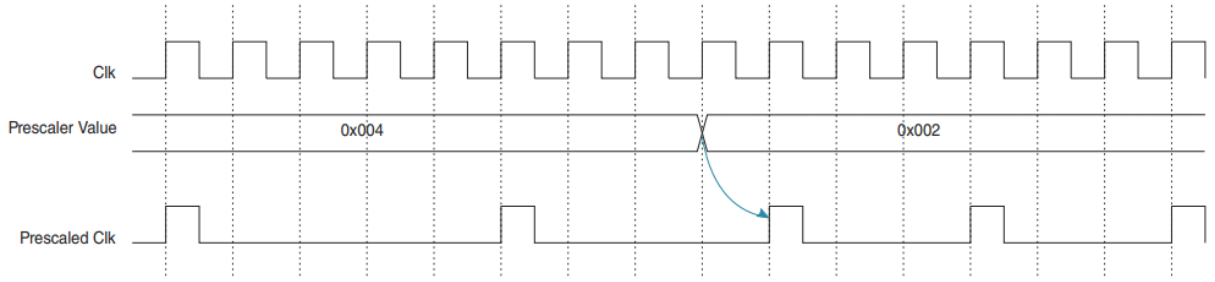


时钟选择(图中没标注可以通过GPT_CLK输入):



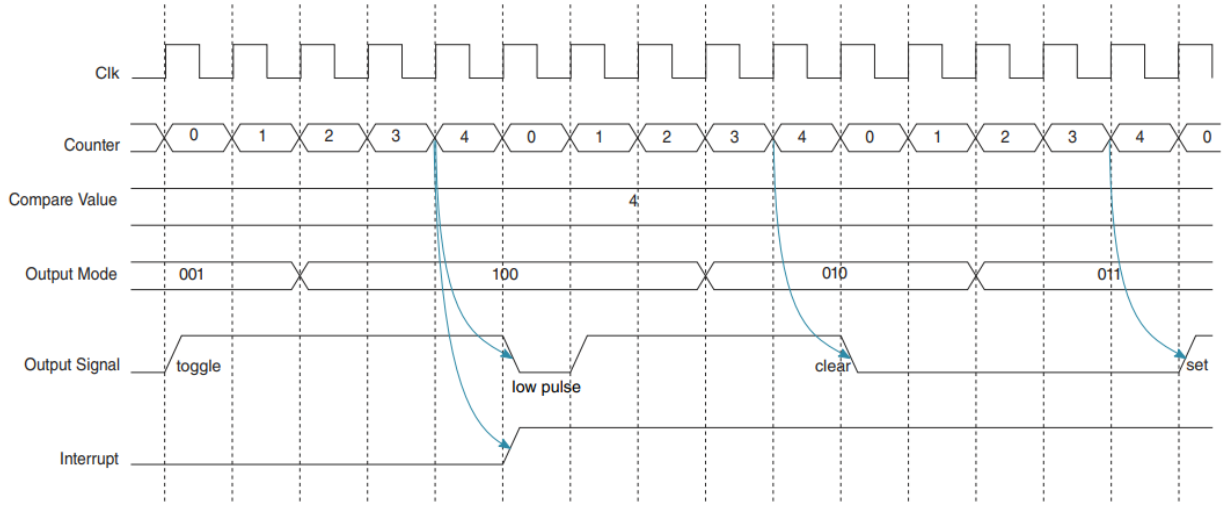
分频修改PRESCALER(范围1-4096,0代表不分频).

时钟选择:需要GPT_CR EN=0,关闭中断,检查SWR,重置计数器,然后重新使能,(其实都短暂暂停了,大概也不是很常用)



计数模式:自动重启(到达比较值归零,写比较值清零,只有比较通道1有这个功能,后面的两条通道比较后也不会归零.),自由运行(到达顶部0xFFFFFFFF归零,相当于没有比较)

输出比较:由模式CR_OM决定.(反转,清楚,设置,一个周期低脉冲)

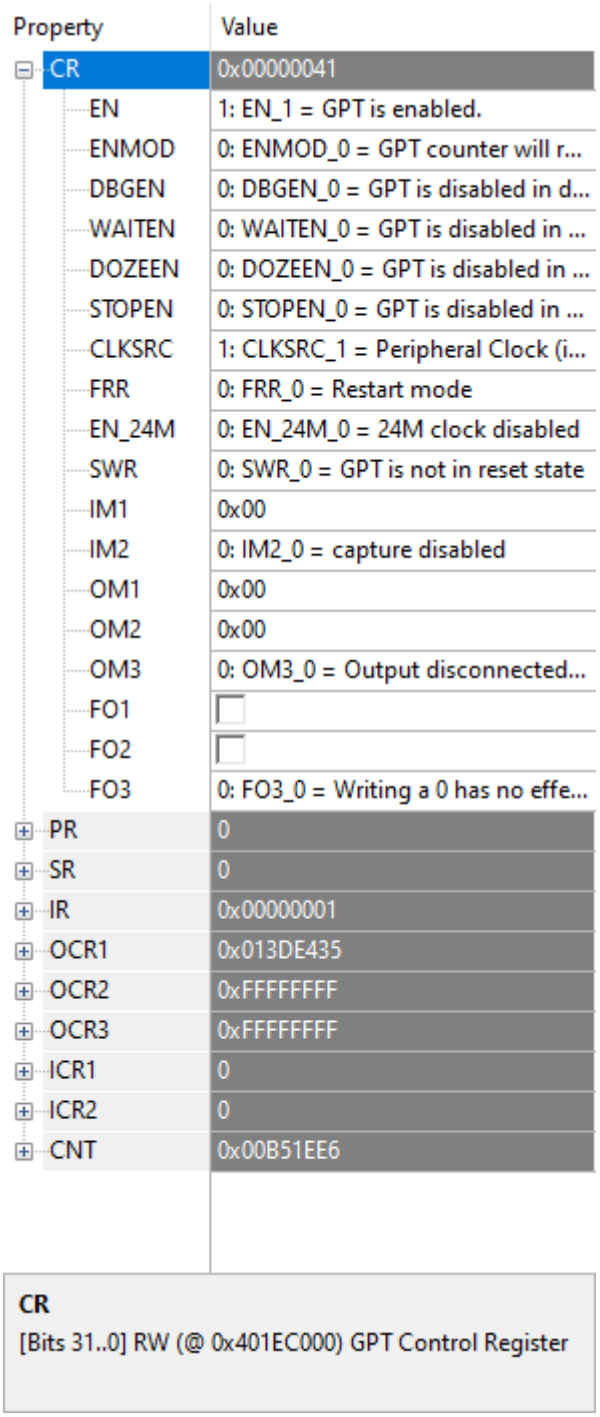


寄存器

看起来比较多,但是实际上功能并不多,用起来也是挺简单的,实际上除了CR寄存器,其他都没什么好说的.

- CR 控制寄存器(只说重点)
  - EN 主使能
  - ENMOD 去使能时CNT是否保留.
  - CLKSRC 时钟源
  - EN_24M 使用24MHz时钟源特殊选项
  - SWR 复位模块
  - FRR 重启模式/自由计数模式
  - IM 捕获方向(上升/下降/上升+下降),只需要使能对应FO,然后配置引脚,对应引脚就捕获波形了,宏定义对应:GPT_CR_IM1(1),每次捕获后加1.

    - FO 输出使能,只需要使能对应FO,然后配置引脚,对应引脚就输出波形了.宏定义对应:GPT_CR_FO1(1),每次输出后执行输出要求的指示.
    - OM 输出模式,和上面FO配合.
- PR 分频寄存器(针对24MHz有额外分频选项)
- SR 状态寄存器(输出比较事件1,2,3,输入比较事件1,2,回归到0计数器)
- IR 中断寄存器(和SR对应,使能对应中断)
- OCRn(n=1,2,3) 输出比较寄存器,存比较数值.
- ICRn(n=1,2) 输入比较寄存器,存比较数值.
- CNT 当前计数



Property	Value
CR	0x00000041
EN	1: EN_1 = GPT is enabled.
ENMOD	0: ENMOD_0 = GPT counter will r...
DBGEN	0: DBGEN_0 = GPT is disabled in d...
WAITEN	0: WAITEN_0 = GPT is disabled in ...
DOZEEN	0: DOZEEN_0 = GPT is disabled in ...
STOPEN	0: STOPEN_0 = GPT is disabled in ...
CLKSRC	1: CLKSRC_1 = Peripheral Clock (i...
FRR	0: FRR_0 = Restart mode
EN_24M	0: EN_24M_0 = 24M clock disabled
SWR	0: SWR_0 = GPT is not in reset state
IM1	0x00
IM2	0: IM2_0 = capture disabled
OM1	0x00
OM2	0x00
OM3	0: OM3_0 = Output disconnected...
FO1	
FO2	
FO3	0: FO3_0 = Writing a 0 has no effe...
PR	0
SR	0
IR	0x00000001
OCR1	0x013DE435
OCR2	0xFFFFFFFF
OCR3	0xFFFFFFFF
ICR1	0
ICR2	0
CNT	0x00B51EE6

CR
[Bits 31..0] RW (@ 0x401EC000) GPT Control Register

相关引脚

示例实现

- 比较中断:可以选择不同的时钟源,24MHz时钟的配置和其他不一样,需要特别注意.
- 时钟源最好一开始就固定好,毕竟一般情况也没那么变态需要老是变更时钟源的情况.
- 芯片上有GPT1和GPT2,配置和使用完全一样.

```
1
2  volatile uint32_t ulCount = 0;
3
4  void GPT1_IRQHandler(void)
5  {
6    /* Clear interrupt flag.*/
7    GPT1->SR = kGPT_OutputCompare1Flag;
8
9    ulCount++;
10   /* Add for ARM errata 838869, affects Cortex-M4, Cortex-M4F, Cortex-M
7, Cortex-M7F Store immediate overlapping
11   exception return operation might vector to incorrect interrupt */
12   __DSB();
13 }
14
15 static void MainTask(void *pvParameters)
16 {
17   /* Initialize GPT module */
18   CLOCK_EnableClock(kCLOCK_Gpt1);
```

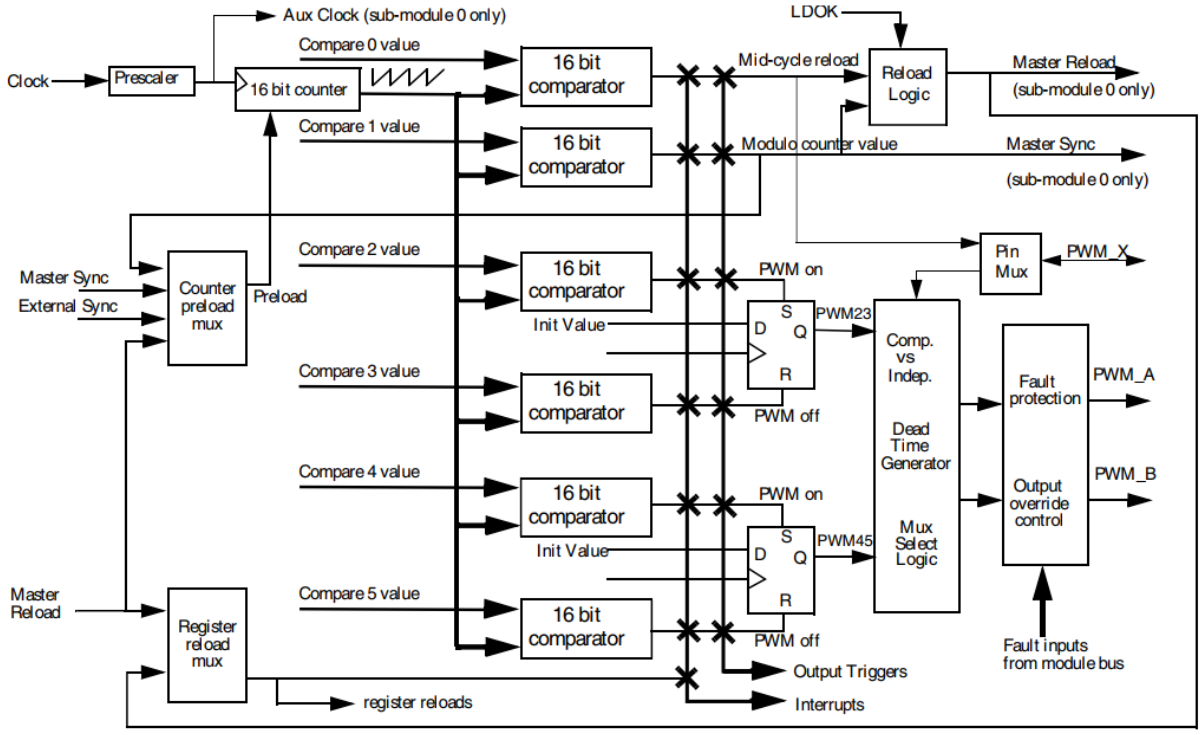
```
19
20    GPT1->CR |= GPT_CR_SWR_MASK;
21    /* Wait reset finished. */
22    while ((GPT1->CR & GPT_CR_SWR_MASK) == GPT_CR_SWR_MASK)
23    {
24    }
25
26    GPT1->CR = GPT_CR_WAITEN(1) | GPT_CR_STOPEN(1) | GPT_CR_ENMOD(1);
27
28    // GPT1->CR = (GPT1->CR & GPT_CR_CLKSRC(0)) | GPT_CR_EN_24M(1) | GPT_C
R_CLKSRC(kGPT_ClockSource_Osc);
29    GPT1->CR = (GPT1->CR & (GPT_CR_CLKSRC(0) | GPT_CR_EN_24M(0))) | GPT_CR
_CLKSRC(kGPT_ClockSource_Periph);
30    // GPT1->CR = (GPT1->CR & (GPT_CR_CLKSRC(0) | GPT_CR_EN_24M(0))) | GPT
_CR_CLKSRC(kGPT_ClockSource_HighFreq);
31    // GPT1->CR = (GPT1->CR & (GPT_CR_CLKSRC(0) | GPT_CR_EN_24M(0))) | GPT
_CR_CLKSRC(kGPT_ClockSource_Ext);
32    // GPT1->CR = (GPT1->CR & (GPT_CR_CLKSRC(0) | GPT_CR_EN_24M(0))) | GPT
_CR_CLKSRC(kGPT_ClockSource_LowFreq);
33
34    /* Divide GPT clock source frequency by 3 inside GPT module */
35    GPT1->PR = GPT_PR_PRESCALER(0);
36
37    /* Set both GPT modules to 1 second duration */
38    GPT1->OCR[0] = CLOCK_GetFreq(kCLOCK_PerClk) / 3;
39
40    /* Enable GPT Output Compare1 interrupt */
41    GPT1->IR |= kGPT_OutputCompare1InterruptEnable;
42
43    /* Enable at the Interrupt */
44    EnableIRQ(GPT1_IRQn);
45
46    /* Start Timer */
47    GPT1->CR |= GPT_CR_EN(1);
48    for (;;)
49    {
50      vTaskDelay(pdMS_TO_TICKS(1000));
51      ulCount++;
52    }
53 }
```

Enhanced Flex Pulse Width Modulator (eFlexPWM)

这个也是个比较复杂的玩意了,而且光有理论不熟悉电机也很难用,而且看名字这么高级,还带增强,肯定不是小东西.框图上元素也是特别多,而且官方文档介绍其他定时器一般只有20页,这里足足100多页,但是无奈官方板上PWM竟然没引出来,IMXRT1011又因为引脚太少(LQFP80),PWM几乎残疾:

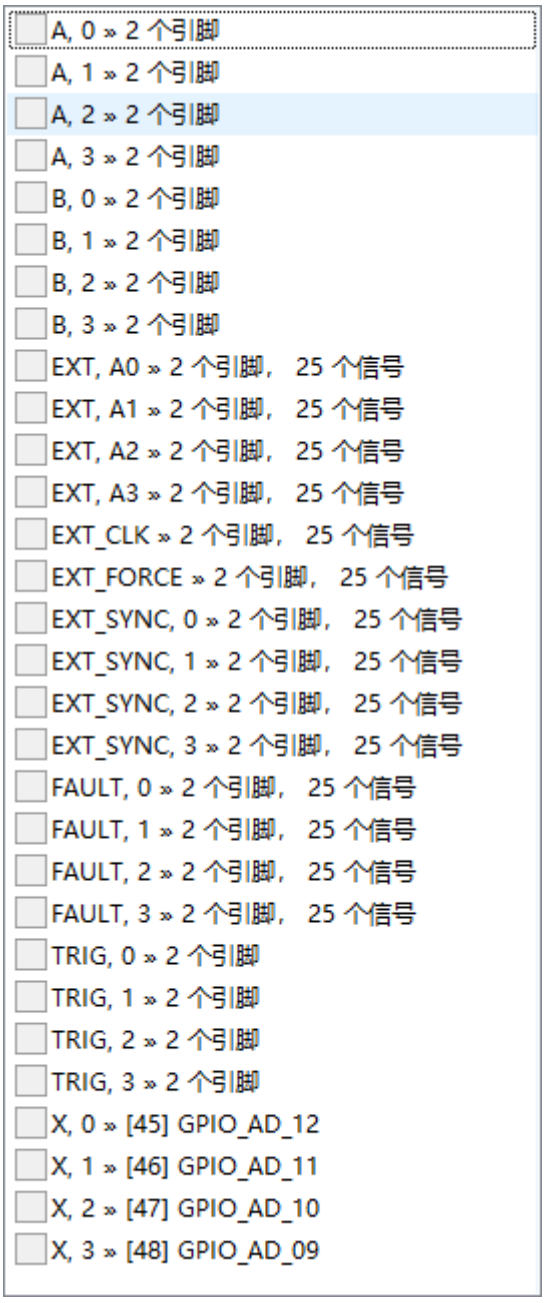
- 16位分辨率,但是有小数PWM时钟(NanoEdge)
- 中心对齐PWM(依据VAL0切开对称),边缘对齐PWM(起点一样,终点不同),相移PWM(相位不同),双开关PWM(SPLIT控制,一个周期两次跳变),非对称PWM(独立发生).
- 互补输出/独立输出,PWMX永远独立.
- 能够接受有符号数的PWM数值(就是可以输入负数啦~,因为他也识别补码.)
- 每个PWM可以独立控制边缘
- 支持和外部硬件同步
- 支持和其他PWM同步
- 双缓冲寄存器(影子装载)
    - 重装频率从1到16
    - 半循环重装
- 每个 PWM 周期可通过硬件生成多个输出触发事件(ADC触发什么也可以)
- 故障输入可连接到多个PWM输出
- 故障输入硬件滤波
- PWM输出极性可独立编程
- 死区时间可独立编程(上下死区都可以分开设定)
- 互补对的死区时间也可以独立编程
- 每个PWM可以独立控制
- 软件上可以控制FORCE_OUT以便同时输出
- PWM_X引脚可以选择从每个子模块输出第三个 PWM 信号
- 支持输入捕获
- 支持比较输出
- 增强的双边沿捕获功能
- 以下信号可以为互补PWM提供触发源:
    - XBAR模块输出(芯片内部的另一个模块)
    - ADC模块输出(考虑ADC看门狗设置的高限/低限)

- 调试时可以暂停输出.
- PWM23/PWM45是内部的,PWMA/PWMB是外部的,而且PWM23和PWM45的控制不一定就等于PWMA/PWMB,因为PWMA/PWMB可以取反输出.
- 一个子模块有4个下面这个玩意.



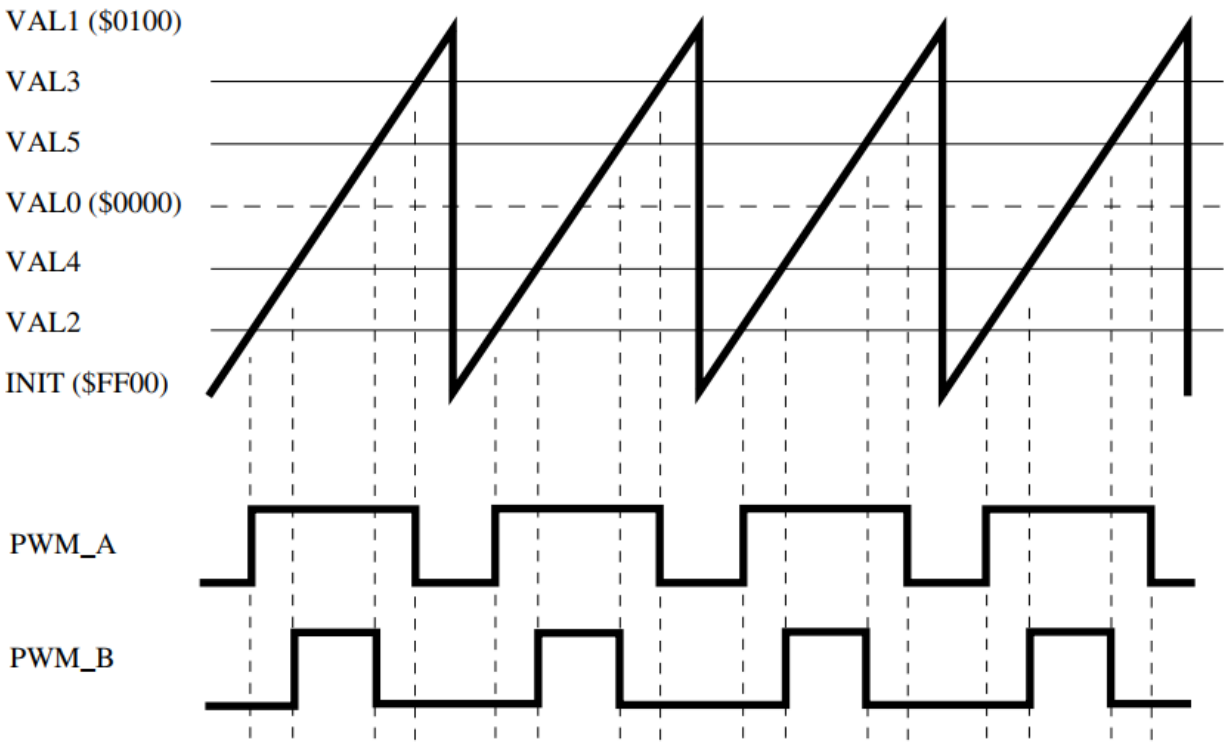
可以用于PWM的信号引脚非常之多,感觉芯片引脚都不够用的那种.

- A,B引脚就是输出对.
- X引脚是辅助引脚,可以输出独立PWM也可以输入进行一些检测操作.
- FAULT急停输入,一旦有效,PWM立马停止输出.
- EXT_*
    - EXT_SYNC 外部同步信号,有效后开始进行PWM计数.
    - EXT_FORCE 外部信号强制更新PWM用途(公共信号,只有一个)
    - EXT_EXTA/EXT_EXTB 外部的备用控制信号
    - EXT_CLK 外部时钟信号(公共信号,只有一个)
- OUT_TRIG 触发输出(也就是下面TRIG引脚)
- 因为有4个子模块,上面说的所有引脚都有4个,比如A0 A1 A2 A3和B0 B1 B2 B3.



PWM模式

中心对齐模式

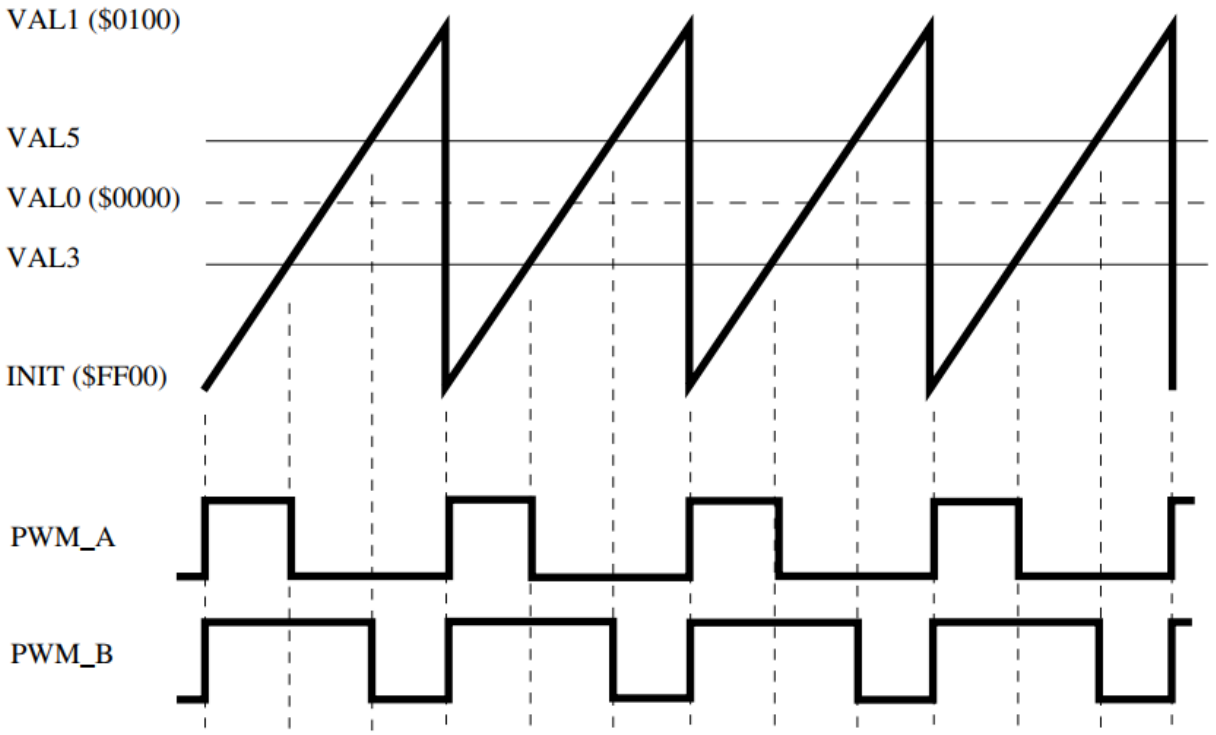


子模块定时器仅在上升方向上计数,然后重置为INIT值,必须指定两个值(打开的数值和关闭的数值),这种双边缘生成不仅可以控制脉冲宽度,还可以控制信号的相对对齐,PWM中心对齐模式本质上是打开和关闭边缘各操作一次.

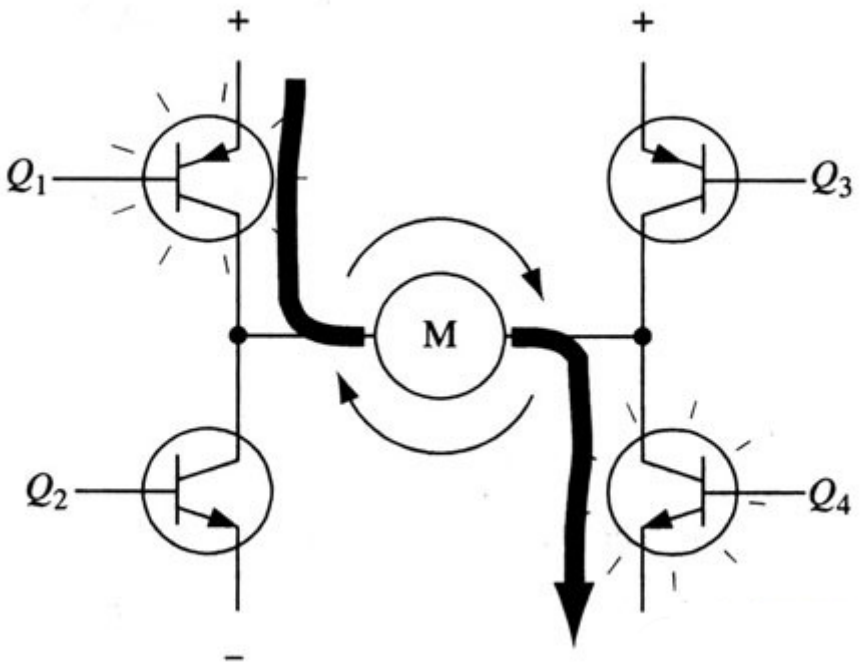
上面的图还说明了PWM生成过程的额外增强,当计数器重置时,将重新加载用户定义的数值,该数值可能为零,也可能不为零,反正你想怎么设定就怎么设定,如果选择的值恰好是带符号状态下的负数(上面的0xFF00就是-256,0x100就是+256,VAL0就是两者相加除以二的数值~),则PWM发生器以有符号模式运行,意味着,如果每个PWM的打开和关闭边缘值也是相同的数字,但它们的符号只有不同,则输出信号的"ON"部分(不一定是高)将围绕零计数值居中,因此,软件中只需要计算一个 PWM 值,然后将该值及其负值分别提供给子模块,作为关闭边缘和打开边缘的数值.

如果所有PWM信号边缘计算遵循同样的惯例(绝对值情况:VAL2-VAL0 = VAL3-VAL0),那么信号将相对于彼此中心对齐(就是上面说的VAL0),这个也是实现目标,当然.信号之间的中心对齐不限于零计数值周围的对称性,只是说用0作对齐的话,比较方便计算.

边缘对齐模式



通过看图观察,看到PWM_A/B都用INIT开始上升,PWM_A受到VAL3控制发生下降沿,PWM_B受VAL5控制发生下降沿,周而复始,这就是对齐INIT边沿,即将每个脉冲的开启边指定为INIT值,产生边对齐操作的结果,只需要定期更新VAL3/VAL5(即关闭边沿)就可以更换脉宽.

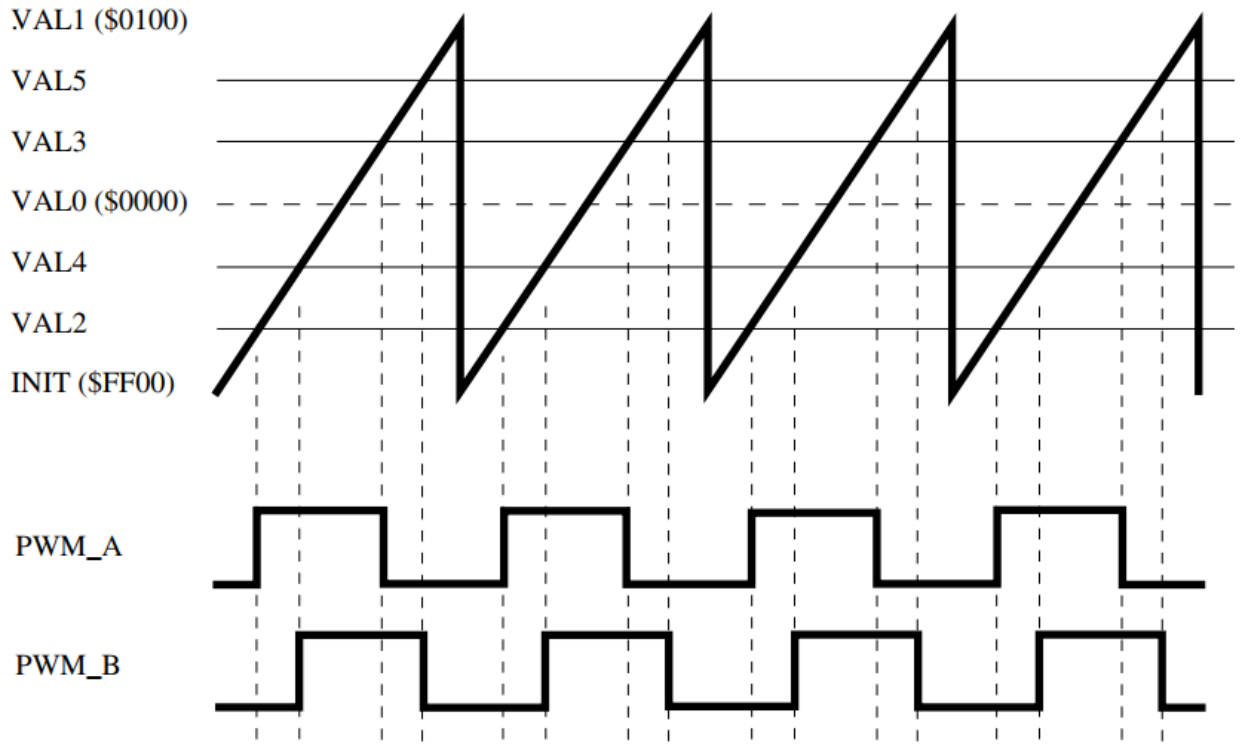


驱动H桥(上图)的常见方法是使用双极性PWM的技术,其中50%占空比会导致负载没有电压(理论就是电机静止了哈, 虽然实际上应该做不到~),占空比小于50%会导致负电压,占空比大于50%的时候产生正电压.如果模块设置为符号模式操作(INIT和VAL1 值是相同的数字与相

反的符号,上图就是这个情况,一个是-256,一个是+256),那么PWM关闭边缘值和电机逆变器电压之间有直接的比例,这样操作起来会比较简单,因为0点正是50%~

上面驱动图里面,Q2=!Q1,Q4=!Q3,Q1=Q4,这是理想情况,所以其实只要一个PWM输入,没有考虑死区,实际情况MOS没那么快反应的啦~

相移PWM模式



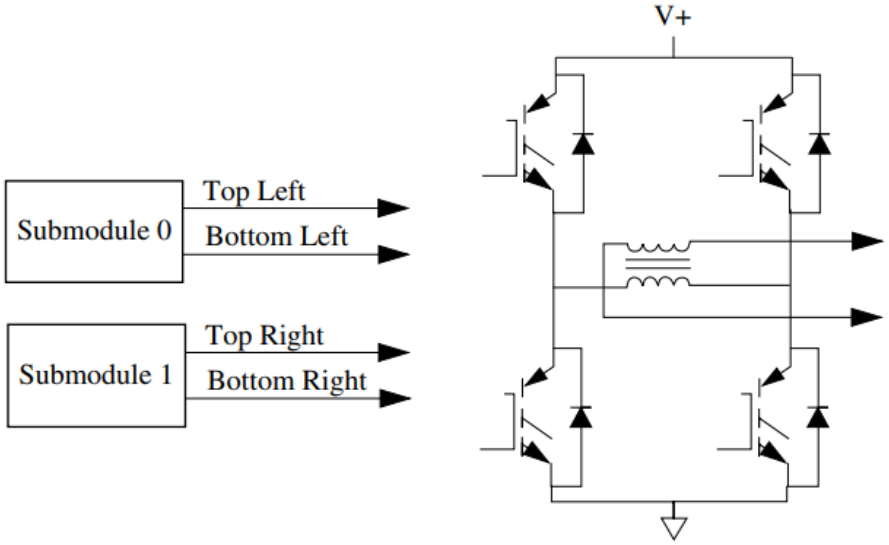
相移就是相位有偏移,但是相同长度的PWM,虽然看起来有点像中心对齐,但是实际上不是同一个东西,相同的是,他们都需要指定开始结束,不同的是中心对齐不会要求两路PWM同等宽度,所以实际上,这里是信号的彼此相移.

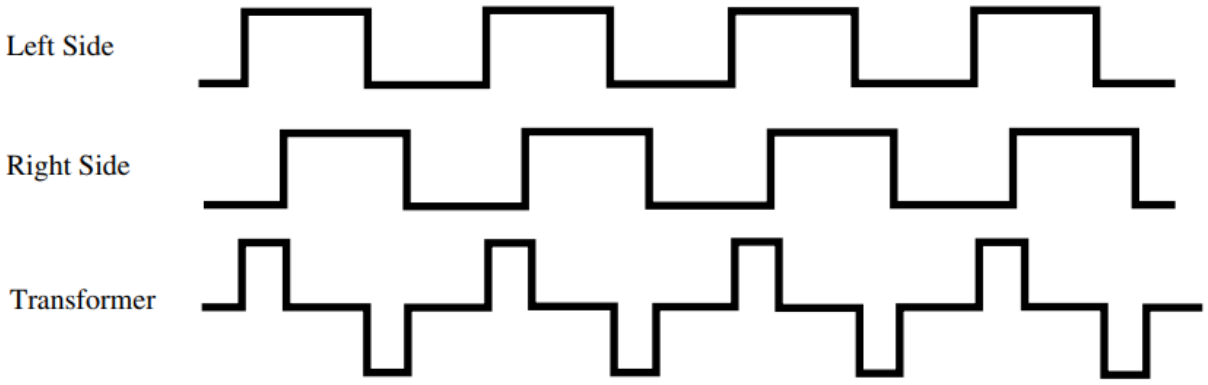
当施加于功率级时,这会产生某些优势.比如说,当以低调制指数操作多相逆变器时,不同相位的所有PWM开关边缘几乎同时发生(所以我们要推迟一些啊,不要同时启动),这样可能产生大量的噪音干扰,可能影响ADC采样之类的.但是,相移不会影响占空比,所以平均负载电压不受影响~

不过呢,既然他是相移,那就有个很简单的方法来设定,比如使用子模块1-3的PHASEDLY寄存器来指示它们的从子模块0时间延迟数值,当选择来自子模块0的主同步信号作为初始化源(CTRL2 [INIT_SEL] =b10)时,可以使用此方法,这种方法允许所有子模块以相同的打开和关闭时间进行编程,并且实施相移.

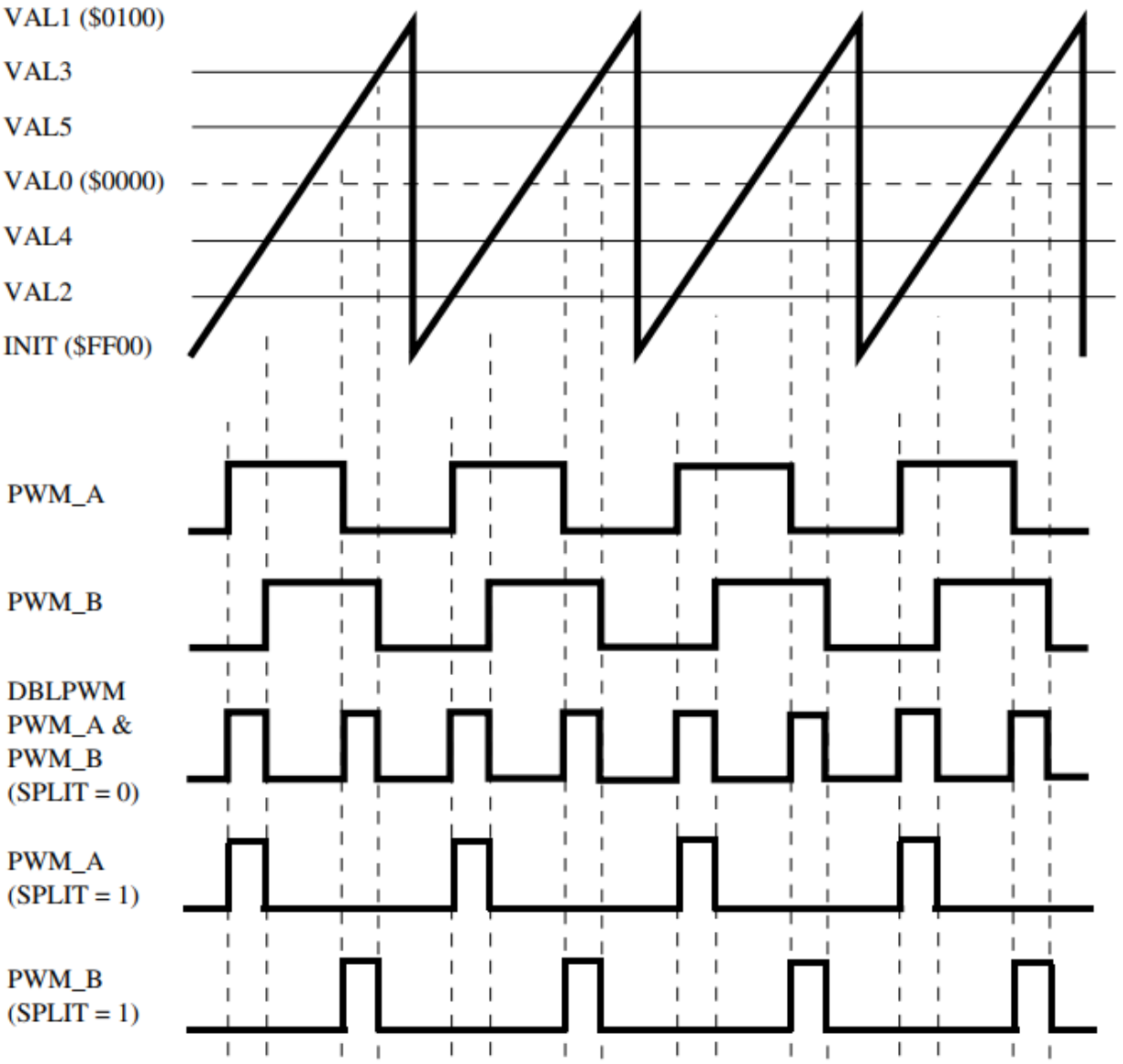
下面显示了相移PWM的额外优势,在这种情况下,H桥电路由2个PWM信号驱动(PWMA/PWMB),以控制变压器主端的电压波形.左侧和右侧 PWM 均配置为始终产生具有50%占空比的方波,这适用于H桥,因为没有产生窄脉冲宽度(这样MOS就不会使劲通断了,甚至可能MOS根本就来不及通断)

H桥右侧的方波与H桥的左侧相比,发生了相移,因此,变压器主模块可以看到其端子上的底部波形,该波形的RMS(均方根电压,即有效值)由方波的相移量直接控制,无论相移如何,只要每个方波的占空比保持在 50%,负载电压中都不会出现直流组件,因此该技术非常适合变压器负载.(不过,虽然举例图是H桥,但是实际上不适用于H桥,因为正负一样,那转个毛啊)



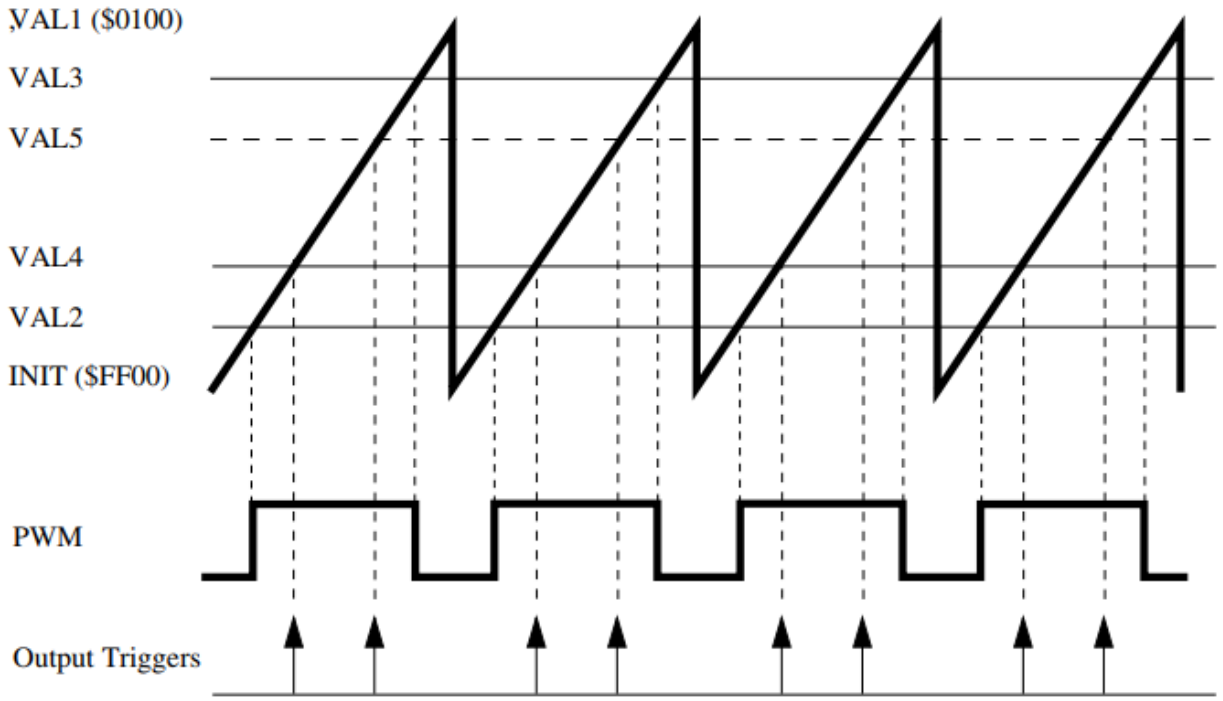


双开关PWM

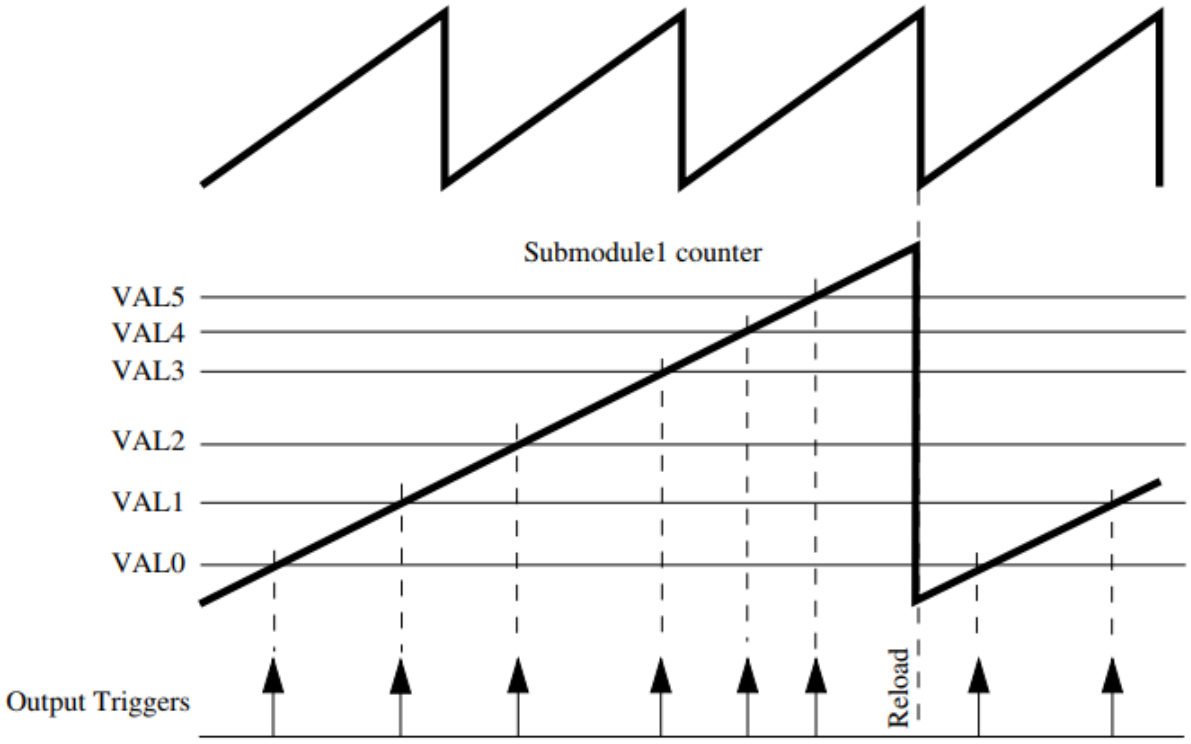


看图就知道这个受DBLPWM控制,看来就是PWM_A和PWM_B的OR之后结果,而SPLIT后是OR之后再砍掉一半结果,这里适用于三相重建,但是操作起来似乎有点复杂.

ADC触发



使用ADC触发,可以更精确地启动ADC,因为这时候用软件ADC会比较不好控制,在每个PWM的变化中,都可以提供触发输出,无需另一个定时器模块.比如设置比较中,VAL2,VAL3用作变化电平但是VAL4,VAL5用作触发ADC采样.



当然你愿意在每个周期输出触发也是没问题的,不过实际上还要考虑ADC转换需要花费时间,这是另外的东西了,这里就先不讨论了(不然就不知道要说到什么时候了~)

增强捕获

增强捕获,实际上就几个简单的功能:

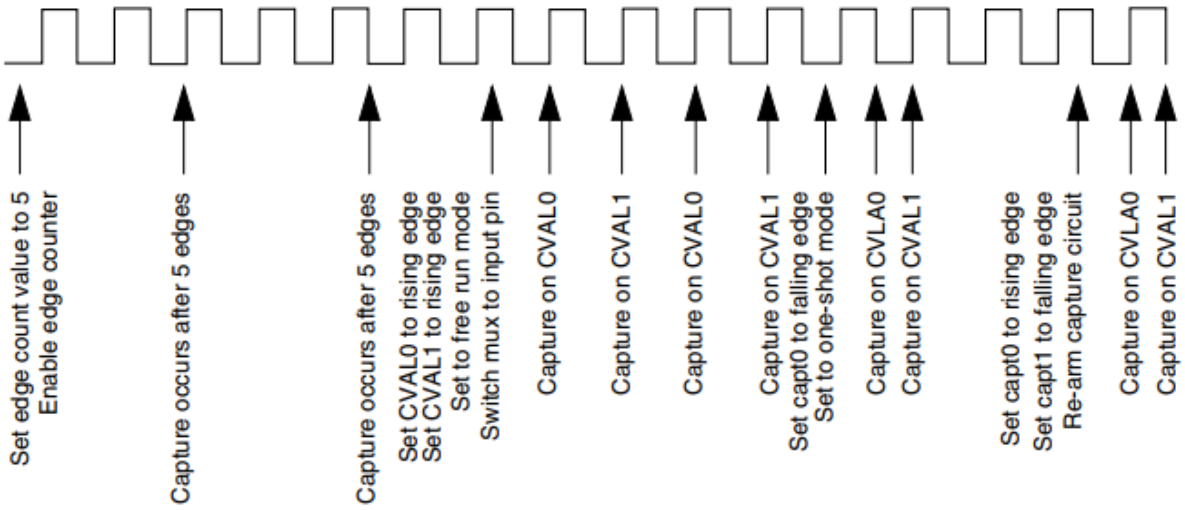
- 捕获输入边沿(用两个比较寄存器)

- 计算两个比较寄存器数值能得知脉宽和周期.
- 可编程的EDGCMP寄存器,用于比较(触发N次后发生事件).

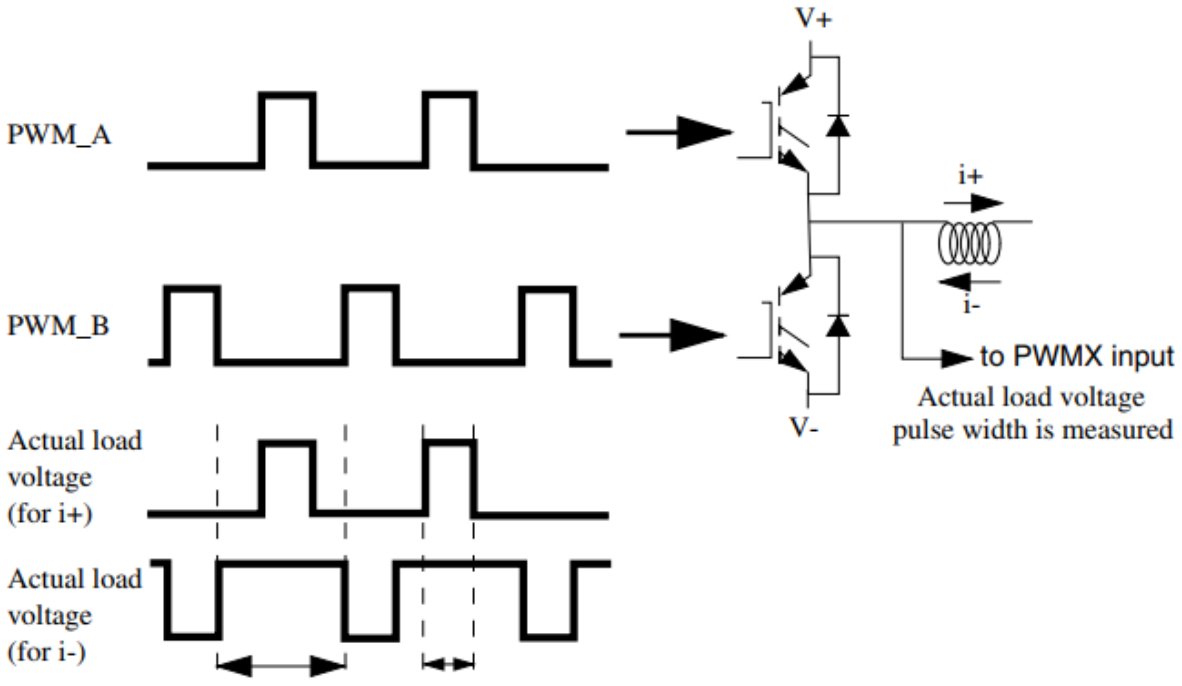
如果PWM引脚不用于PWM功能的时候,就可以用作输入捕获功能,其实输入捕获也是抓取PWM波形的一个方法罢了,没那么神秘啦~,现在我们回想一下,对于PWM产生,PWM 信号的两个边缘通过单独的比较寄存器值来指定(这样就可以在比较匹配时候想做什么就做什么了,比如一个比较设置低电平,一个比较设置高电平),但是当为输入捕获进行编程时,这两个寄存器在同一个引脚上工作,以捕获多个边(因为有上升沿和下降沿边),并以自由运行或一次性方式从一个边切换到另一个边,只需简单地编程每个捕获电路所需的边缘,即可轻松测量输入信号的周期和脉冲宽度(两个捕获寄存器的差值),另外,输入信号的每个边缘都可以设置一个8位计数器,其中计数器输出与用户指定的值 (EDGCMP)进行比较,当计数器输出等于EDGCMP时,将捕获子模块定时器的数值储存并重置计数器,这个功能允许模块对指定数量的边缘事件进行计数,然后执行捕获和中断.

从左往右事件:

- 将边缘计数器数值设置为5,然后使能边缘计数器.
- 经过5个边缘后,发生捕获事件.
- 再经过5个边缘,又发生捕获事件.
- 将CVAL0设置为上升沿捕获,将CVAL1设置为上升沿捕获,设置成自由运行模式,设置MUX.
- 接着的第一个上升沿,捕获到CVAL0.
- 再接着的第二个上升沿,捕获到CAVL1.(其实这个时候计算CAVL1-CAVL0就可以推算周期频率,但是单脉宽还不行,毕竟不一定是50%信号.)
- 再再接着的第三个上升沿,又捕获到CAVL0.
- 再再再接着的第四个上升沿,又捕获到CAVL1.(因为是自由运行模式,所以会一直触发)
- 现在把CAVL0设置成下降沿捕获,并修改为只捕获一次就停模式(ONE-SHOT)
- 紧接着的下降沿就存到CAVL0
- 紧接着的上升沿存到CAVL1(现在周期频率脉宽都可以了,所以测量脉宽条件是上升下降都捕获.)
- 因为是ONE-SHOT,所以后续的上升沿下降沿都不会发生.
- 设置CAVL0上升沿捕获,CAVL1下降沿捕获,接着又重复上一次那样的触发了~(这里体现自由度还是挺高的)



在下面的例子,PWM功率级的输出连接到PWM_X引脚(而PWM_A,PWM_B用作控制输入).PWM_X当前配置为自由运行的输入捕获,并设置CVAL0捕获电路针对上升沿进行编程,CVAL1捕获电路则针对下降沿进行编程,这样可以不断获取脉宽啦(CVAL1-CAVL0).一般用于在驱动电感负载的半桥电路上执行死区失真校正,另外,这些值可以直接与负责生成PWM输出的VALn(就是对应的通道比较寄存器)进行比较,这样就知道PWM发生后多久才传递到电机(传播延迟,以及电机的电流是不能瞬间改变的,他会继续保持直到电流削减.)

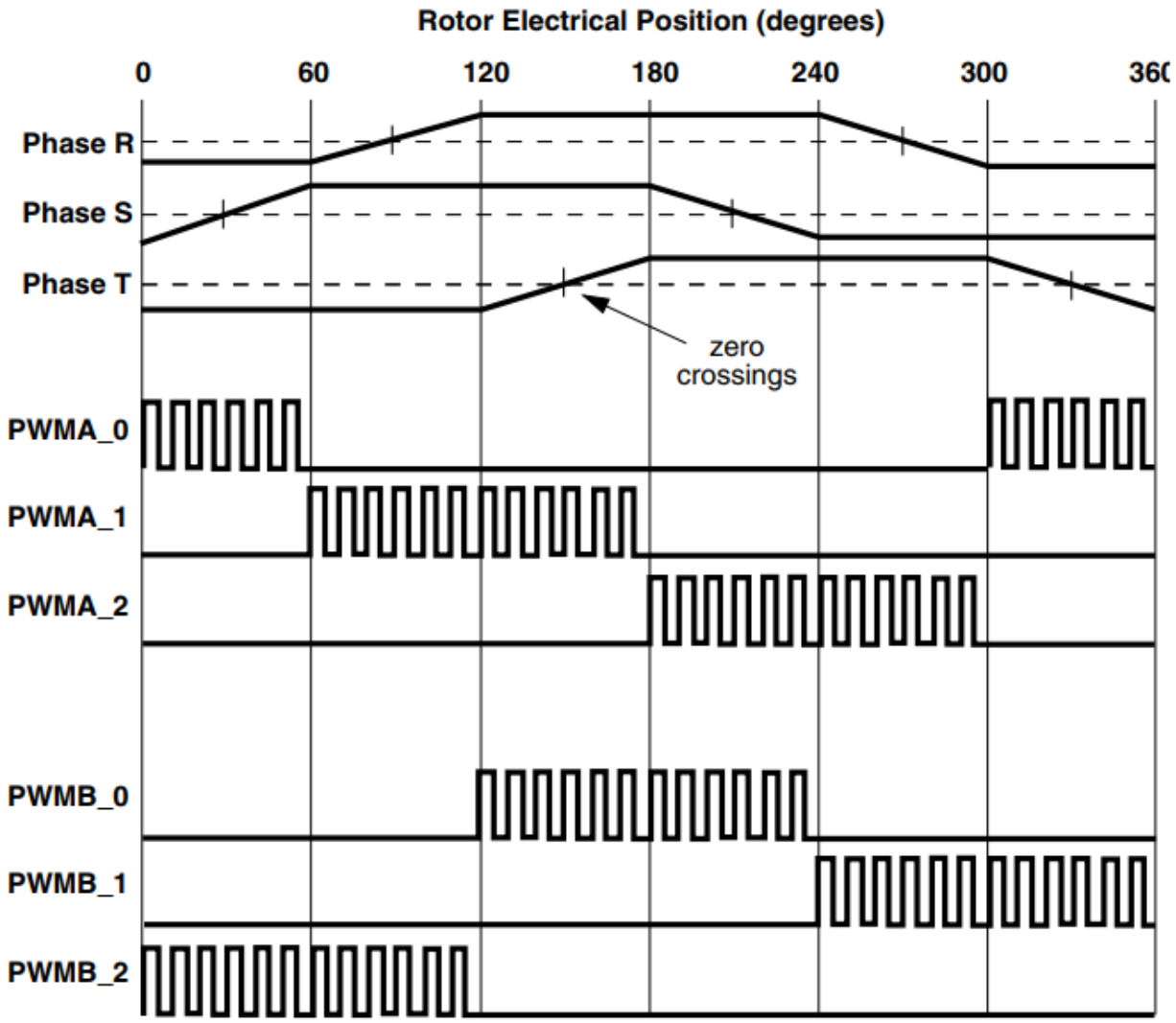


多路输出的同步切换

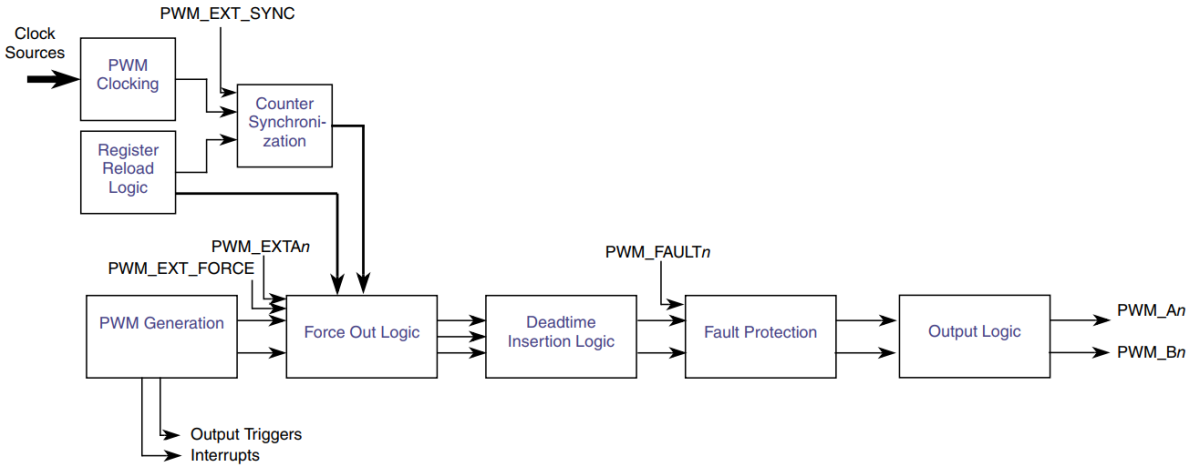
这个功能说白了,就是预先设定好下一步操作,这样到下一步操作时候,由硬件切换,就可以达到零延迟的目的.(有时候零延迟不一定是好事,幸亏有死区自动插入~)

同步输出是通过FORCE_OUT的信号来控制,这个信号可能是寄存器位,也可能是引脚输入(毕竟之前说了那个EXT_FORCE,就是本来芯片引脚就不多还这么奢侈,有一种难以言喻的感觉.)

下面的图说的是一个无刷直流电机(上面是每一相的反电动势),一般情况来说电机需要旋转,但是不需要霍尔传感器,相反,监控电机的反电动势,通过反电动势得到的信息来控制下一次切换.而调节根据过零检测决定,这样PWM数据可以提前配置,当这样配置的时候(并且上一周期完成),定时器的输出比较会驱动FORCE_OUT信号,立即将PWM引脚的状态更改为下一个换向状态,所以无软件延迟.(就是下面的图,60度,立即切换,120度,又立即切换~)



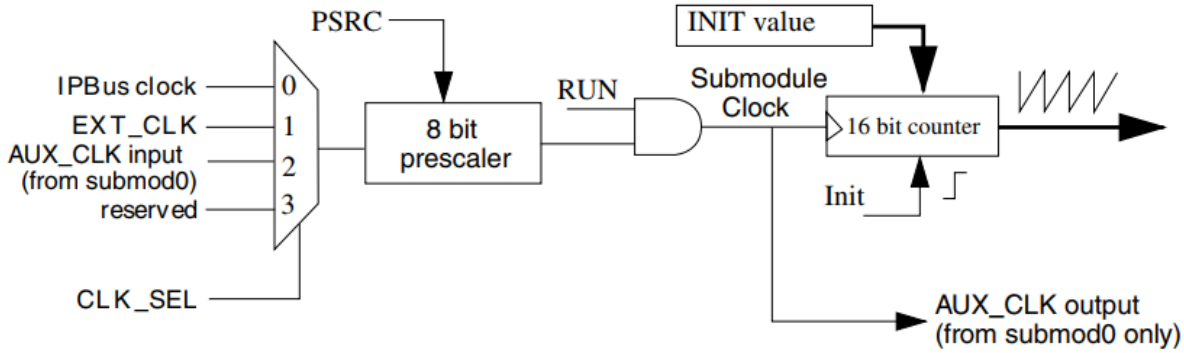
功能细节总体框图



PWM时钟

每个子模块可以在三个时钟信号之间进行选择:

- IPBus 时钟
- EXT_CLK
- AUX_CLK(AUX_CLK是从子模块0输出得来的,并且可以由其他子模块选择这个时钟作为时钟源输入,以便来自子模块0的8位预分频器和MCTRL[RUN]控制所有子模块~)



为了达到降低PWM频率的目的(通常就是因为上级时钟太快了~),所以后面还放了个预分频,分频系数从1(不分频)-128可以选,这个设计很常见,要让预分频器有效,需要设置MCTRL[LDOK](加载寄存器)并开始新的PWM周期或者设置CTRL[LDMOD](立即加载).

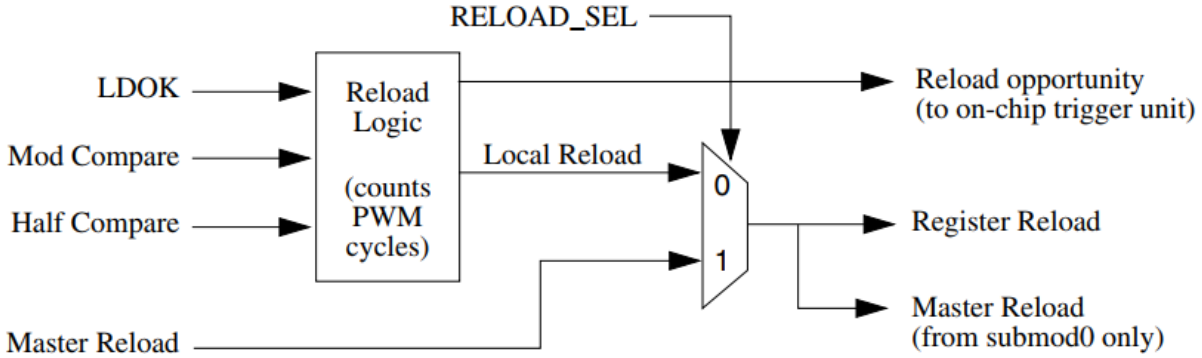
最后上面图说的RUN标志,这个标志一旦置位,就会让CNT接受时钟,就好了~

寄存器重载逻辑

这个概念就像影子,孪生这些概念差不多.寄存器不是写入后立即就有效的.他有一个重载逻辑,这个重载逻辑控制寄存器什么时候有效.

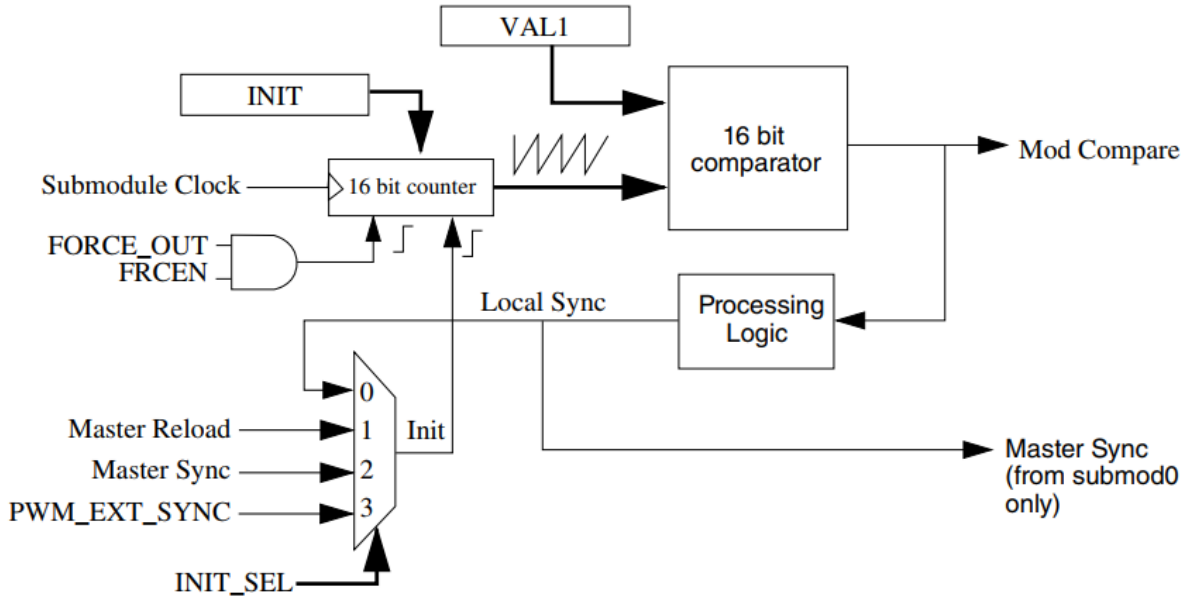
控制寄存器中CTRL[LDFQ]和CTRL[FULL/HALF],可以设置让实际寄存器在n个PWM后进行重载,而CTRL[FULL/HALF]决定重载发生在PWM周期内还是PWM周期结束的时间,如果设置了HALF,就是看VAL0,决定什么时候重载(不一定要中间).

来自子模块0的重载信号可以作为主重新加载信号进行广播,这样所有重载可以全部同步起来.(但不代表定时器同步~)



定时器同步

下面的图说明了定时器的计算逻辑,有一个16Bit的比较器,他的第一个输入是定时器,另一个输入是VAL1(就是比较数值),如果VAL1和定时器输入相同,这样就会输出一个信号.这个信号又会输出到下面化出的处理逻辑(Processing Logic),然后处理逻辑会做什么东西,又是INIT_SEL控制,然后INIT_SEL又可以选择本地同步(这个信号用作计数器的初始化信号),然后又回传到计数器的逻辑块里面,从而控制定时器的周期(因为周期性地重新初始化计数器~).



主同步信号起源为来自子模块0的本地同步,如果配置为这样做,则任何子模块的定时器周期都可以锁定为子模块中的计时器周期(因为子模块0到点复位主计数器~)

PWM_EXT_SYNC就是外部同步的意思了(可能是外设外,也可能是芯片外引脚上,具体取决于系统架构),这个信号可以选择作为计数器初始化的源(上面INIT_SEL写的辣么明白),这样就可以通过PWM_EXT_SYNC控制所有子模块的周期(之前只是用子模块0做复位,现在外面用作复位一样是可以的~)

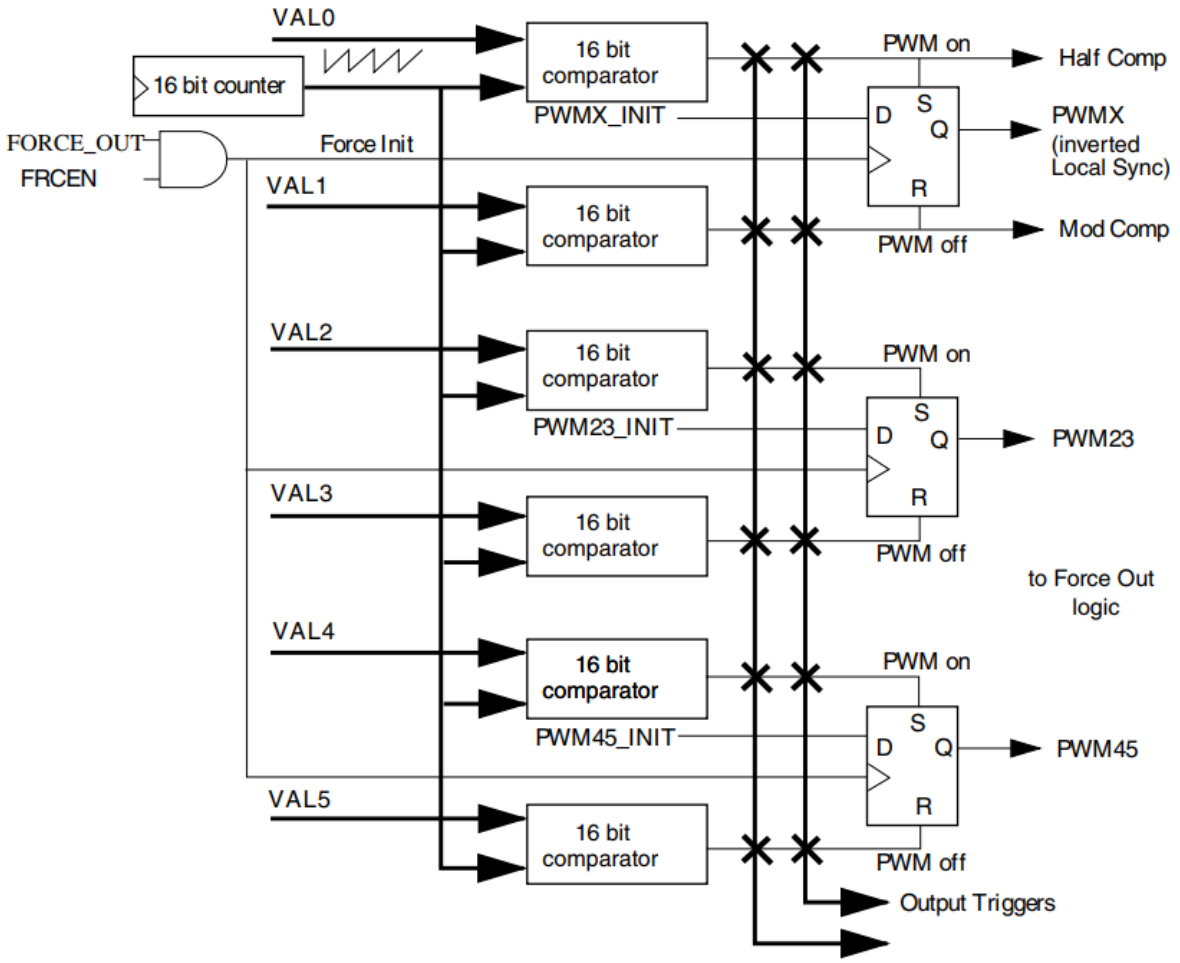
如果主重载信号(Master Reload)选为计数器初始化的源,但是主重载信号又只能从子模块0生成,所以计数器的周期就锁定为子模块0寄存器重载频率,在很多场合下,重载频率通常和软件采样频率相对应,在这个时候,模块的计数器周期等于采样周期,所以,这个定时器也可以用作触发器功能的输出,在一个周期内也可以生成多个PWM(毕竟不止一个子模块).

上面的图还包含了CTRL2[FRCEN](这个寄存器控制FORCE_OUT),如果他可以使能,这个门逻辑就可以通过啊,这样FORCE_OUT也能重载计数器啊.(悲催的计数器,那么多人可以重载他~)

假设,这个时候还加上MCTRL[LDOK]在捣乱,这个初始化信号还会导致所有寄存器发生重载!那么说来,还是FORCE_OUT比较彻底哈,一旦他重载并且满足条件,全部PWM信号都会重启.

PWM生成

下面的图说的是每个子模块自己生成PWM,其中下面输出PWM23和PWM45,而上面只输出PWMX,每个PWM输出信号都是由VAL寄存器比较输出的.(在前面已经说过了,可以一个控制上升沿一个控制下降沿,后面还有个时钟控制RS触发器.)



本地同步信号(就是PWMX下面那行字说的东西~)的生成方式与子模块中的其他PWM信号完全相同(其实就是PWMX信号的非),比较器0会导致本地同步信号的下降沿,而比较器1则生成上升沿,比较器1还与重载逻辑连接(图上似乎没画),以便更准确控制周期.

如果在定时器中,VAL1在控制模块并且满足2VAL0=(VAL1-INIT),则半循环重载脉冲在整个定时器计算周期的一半时候出现,本地同步这时候有50%占空比,

如果不想做同步信号,也可以当普通PWM用,生成一路PWM_X,这样一个模块就有三路PWM了哈!

输出比较

VALx和外部计时器信号都输入到16位比较器,所以可以做输出比较,这一点很容易理解.

- 比较时候输出高
- 比较时候输出低
- 比较时候发生中断
- 比较时候输出触发信号

在比较器后面看到有一个D触发器.D触发器本质是一个记忆逻辑,一个输入D,一个输出Q(其实还有个Q'),当时钟由0转变到1时候,输出会和输入一样,从1变成0之后,D怎么输入,Q都不会

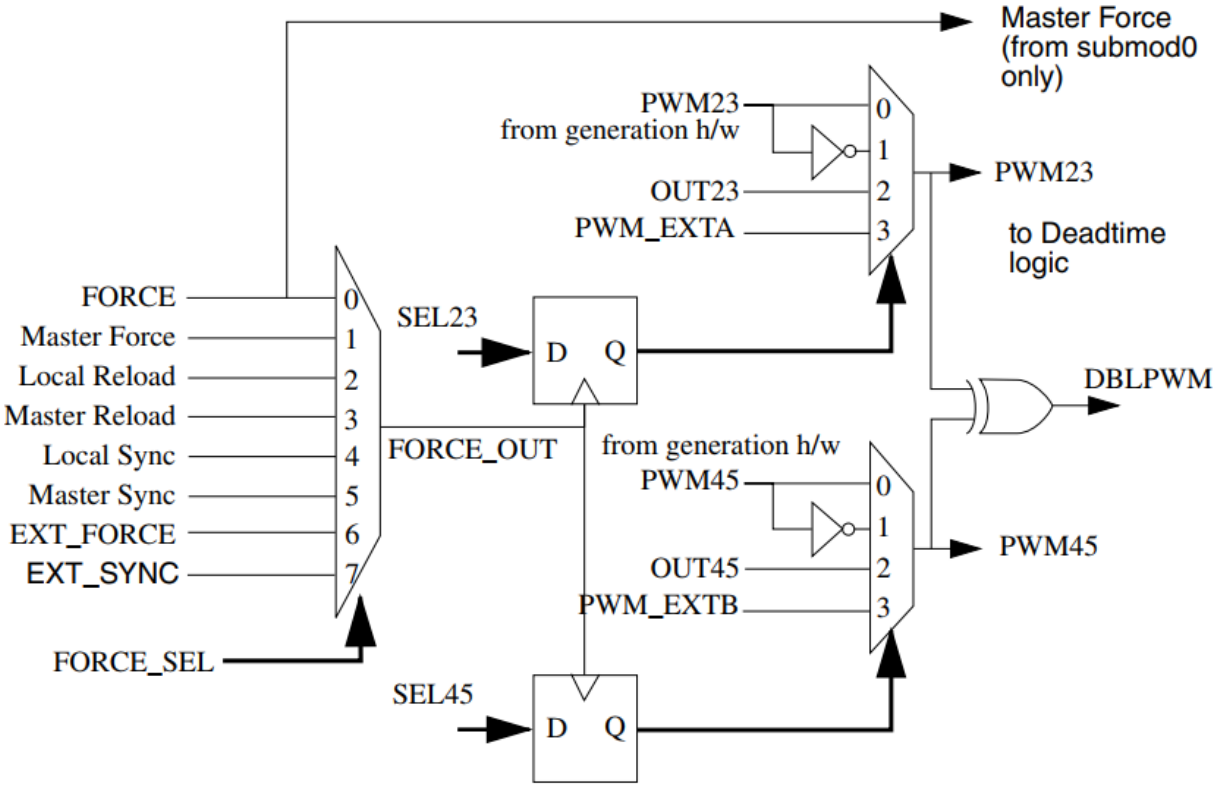
变.S是强迫Q为1,R是强迫Q为0,而这个器件内部的逻辑,实现的本质是强迫S或者强迫R的变化.D是由PWM_INIT控制的,时钟其实不是一个真的脉冲输入,而是FORCE信号(之前已经说到过),既然这里本质是强迫Q变化,那么假设VAL2比较生成,PWMON强迫PWM23变化为高(S),直到VAL3比较生成,PWMOFF生成,才强迫PWM23为低(R),当然这还是没考虑INIT和FORCE情况.

除了这些东西,还看到有个Triggers先串联起来,是因为所有VAL比较,都可以输出事件,这个很好理解.

强制输出逻辑

上面说到的FORCE信号,软件可以在8个信号源中选择FOREC_OUT信号(当然有些芯片并不全),但是如果要求所有子模块输出上的所有信号同时更改,就只能从Master Force,EXT_SYNC.EXT_FORCE选择.

下面还看到SEL23/SEL45控制了一个不知道什么玩意,他们对应控制了PWM23,信号输出,决定他们输出的是正常信号,还是倒置(非)信号,还是由OUT23/OUT45规定的电平,或者是PWM_EXTA/PWM_EXTB规定的外部信号.



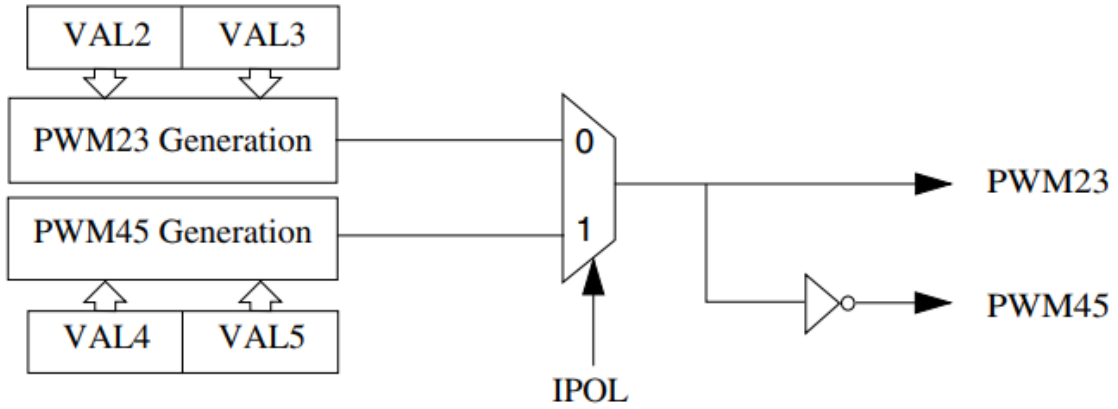
子模块0的本地CTRL2[FORCE]信号可以作为主力信号广播到其他子模块,此功能允许子模块0的 CTRL2[FORCE]同时更新所有子模块输出.

EXT_FORCE信号源自PWM模块外部,但是不一定是外部引脚,比如ADC中的定时器或数字比较器.

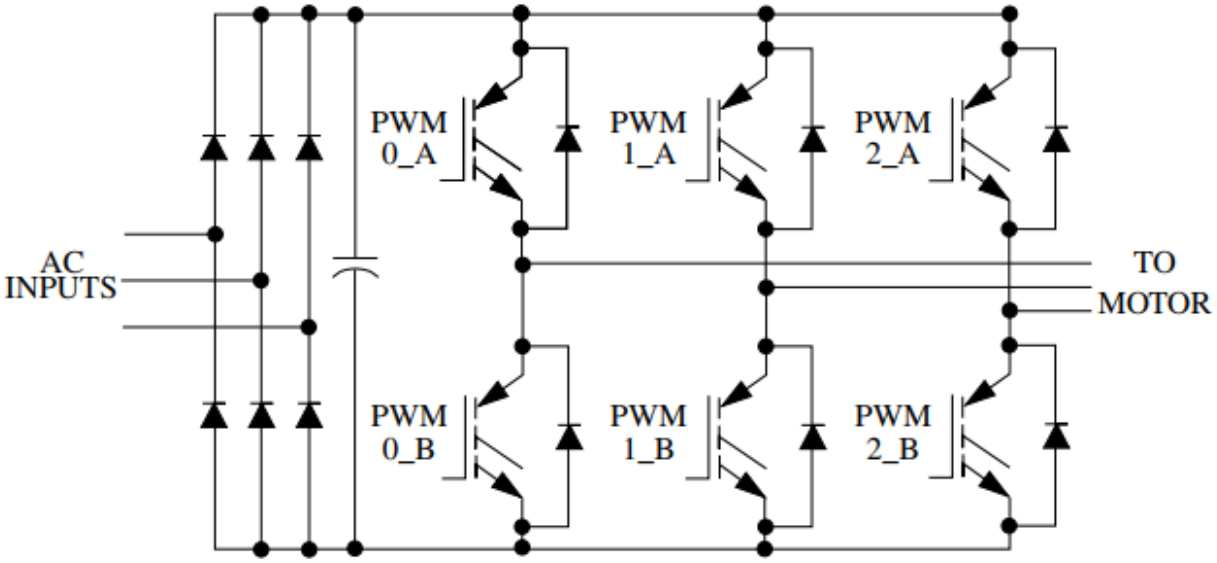
独立或互补通道操作

将1写入CTRL2[INDEP]会将PWM输出配置为两个独立的PWM通道,每个PWM输出由其自己的 VALx控制,独立于另个输出.(唉~ 又可以多生成几路PWM了,只不过是独立的)

将0写入CTRL2[INDEP]会将PWM输出配置为一对互补通道,水作为最终输出由MCTRL[IPOL]决定.

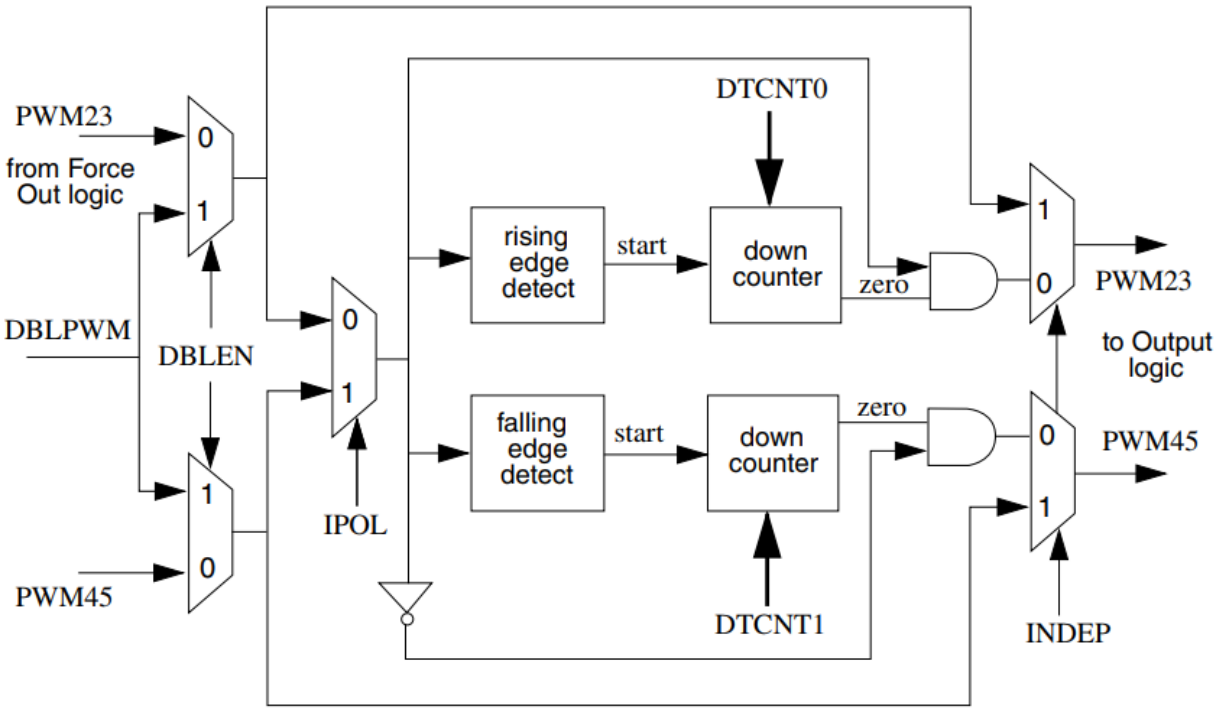


互补通道操作用于驱动电机中的顶部和底部晶体管驱动电路(当然后面还会说死区的问题,因为这里确实需要死区,不然就BOOM了.)



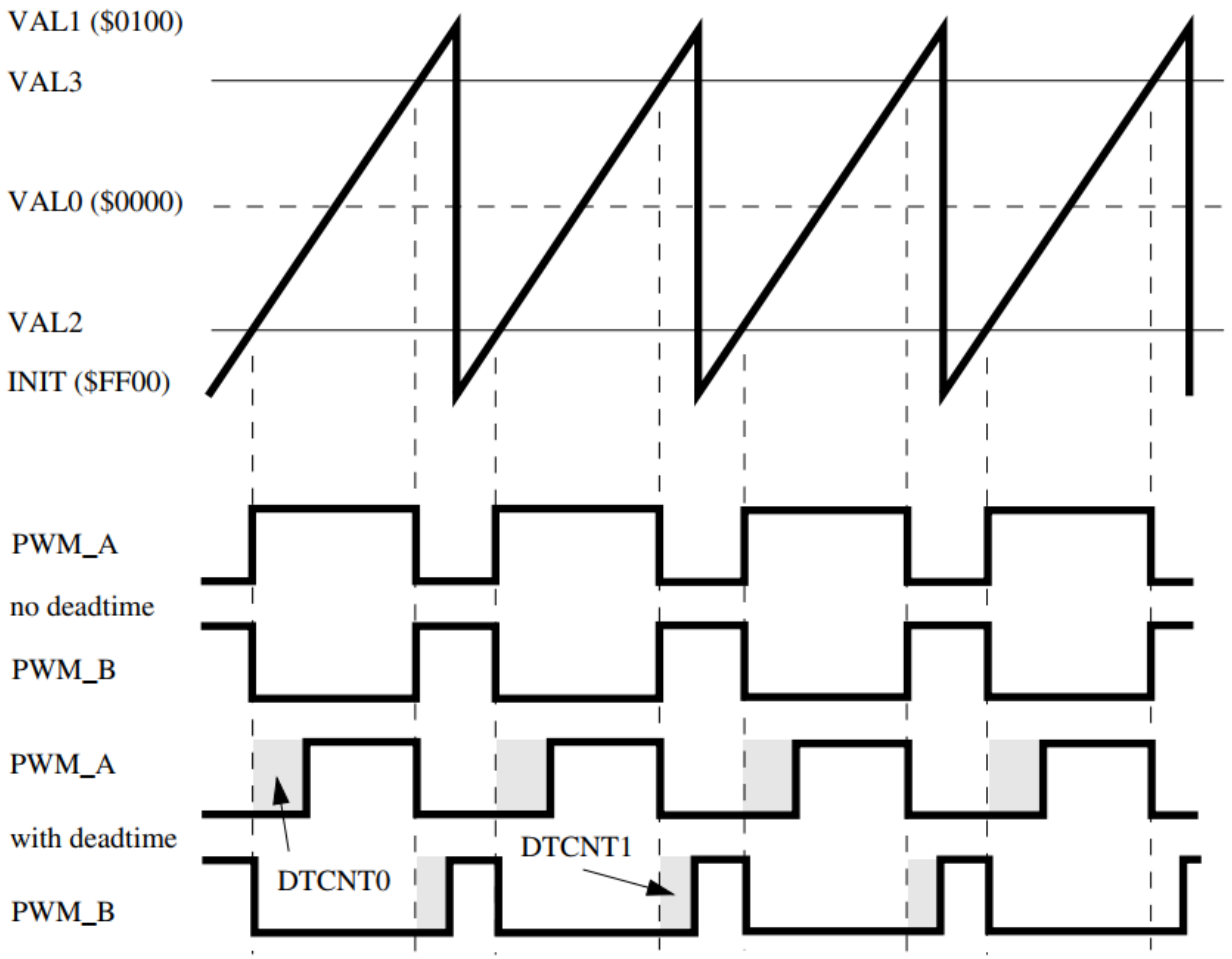
死区插入逻辑

死区只能在互补模式下用,都不互补的话没对应关系怎么插入死区!



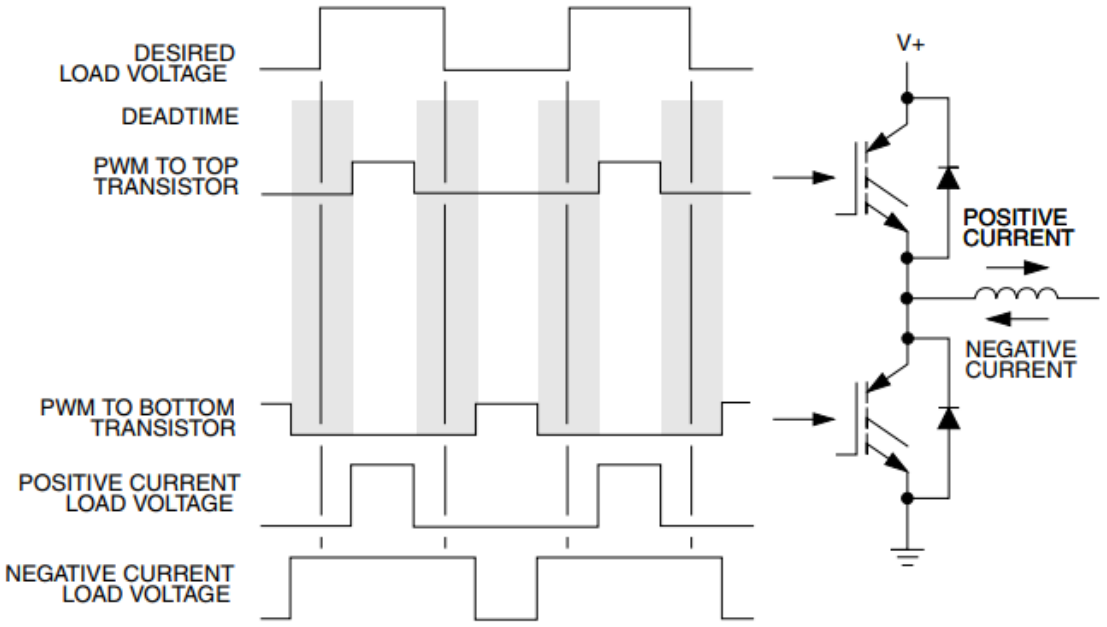
死区发生器就是自动插延迟,下面得图描述很清楚,死区时间寄存器DTCNT0和DTCNT1指定死区有多长(单位:IPBus时钟周期数).

虽然看起来很完美了,但是对于逆变器还是没有办法,当连接到驱动电感负载的逆变器的互补PWM信号中时,逆变器输出上的PWM波形的占空比将与PWM模块输出引脚上的占空比不同,导致施加于负载的电压失真,所以,还有个死区修正的额外功能.



死区顶部/底部修正

死区时候,两个晶体管都是关闭的,所以整个负载中就没电输入了,那输出电压就和负载电流状态有关,并且在输出电压中引入失真(最后就是低速性能差,感觉粗糙.)

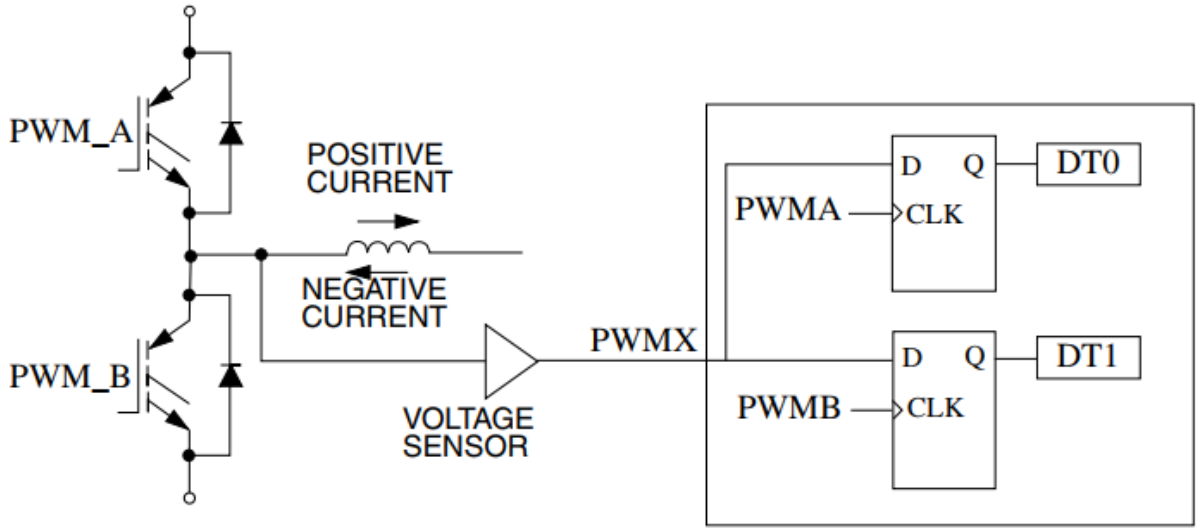


因为插入了死区,所以原来的PWM宽度就会变小,这样所需的功能就和实际计算有出入,另外每个晶体性能(开关延迟)不是完全相同的,更加导致失真,但是这个模块设计了一个功能,可以

给PWM模块提供晶体管信息,这样来纠正失真.(真是傻瓜式)

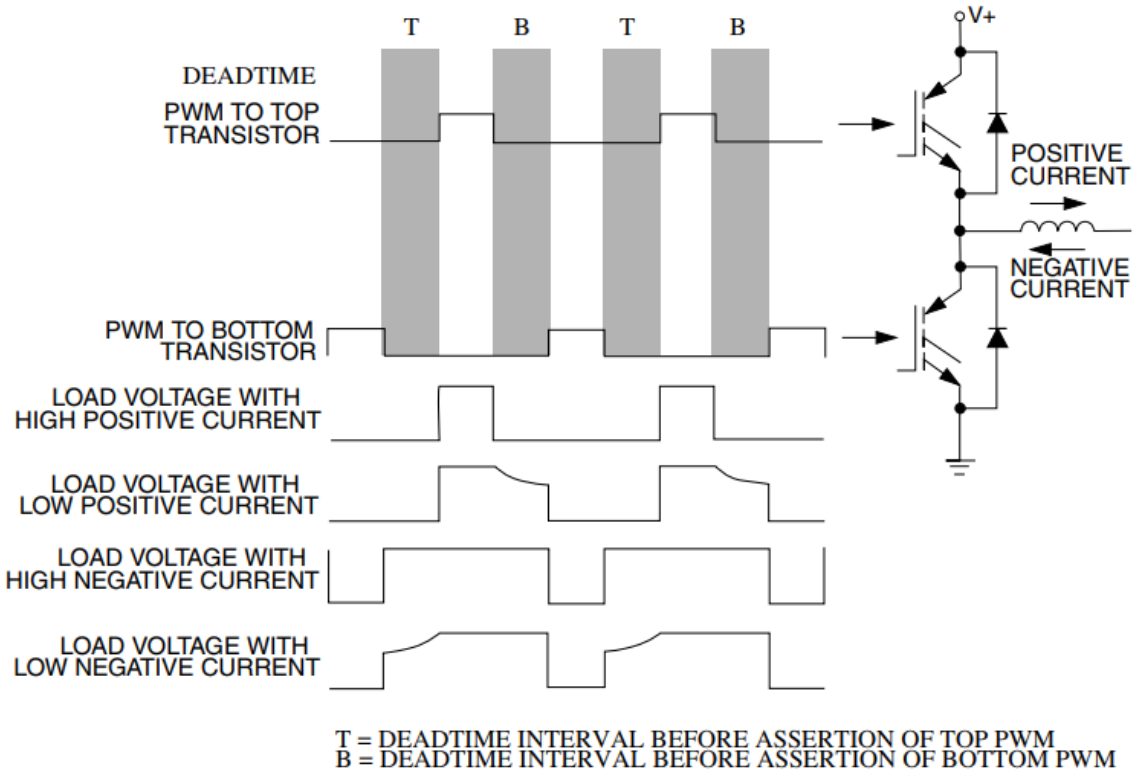
死区手动修正

为了检测电流状态,每个PWM引脚上的电压在每个停机时间结束时采样两次,该值存储在CTRL[DT]中, CTRL[DT]是一种时序标记,特别指示何时在PWM值寄存器之间切换,然后,软件可以将MCTRL[IPL]设置为根据CTRL[DT]值在VAL2/VAL3和VAL4寄存器对之间切换.



如果电流大且流出互补电路,则D触发器锁存器在死区期间均为低电平,即CTRL[DT]=00.如果电流也很大并且流入互补电路,则两个 D 触发器都会在死区期间锁住高电平CTRL[DT]=11.

然而,在低电流条件下,互补电路在死亡时间的输出电压介于高电平和低电平之间,无论极性如何,电流都无法自由地通过反穿二极管,从而在电流穿过零时产生额外的失真,所以采样结果将为CTRL[DT]=10,所以,这个时候更改PWM数值最好了.



分数阶延迟逻辑

看标题就知道这个是为了提高分辨率的,实际作用也是这样,比如如果需要高于IPBus时钟周期的应用,但是不是所有的PWM子模块都支持,得看看他有没有NanoEdge模块,如果有的话,也就改改寄存器的事情.

NanoEdge需要将IPBus时钟设置为定义的频率,置位FRCTRL[FRAC_PU]给模块上电,置位FRCTRL [FRACx_EN]来使能模块.

NanoEdge的最精细粒度是IPBus的1/32,比如要控制一个PWM周期是12.25,那么先用0x000C对VALx编程,然后用0x4000(对于0xFFFF + 1他是1/4)对FRAXVALx进行编程,这样就得到12.25了.

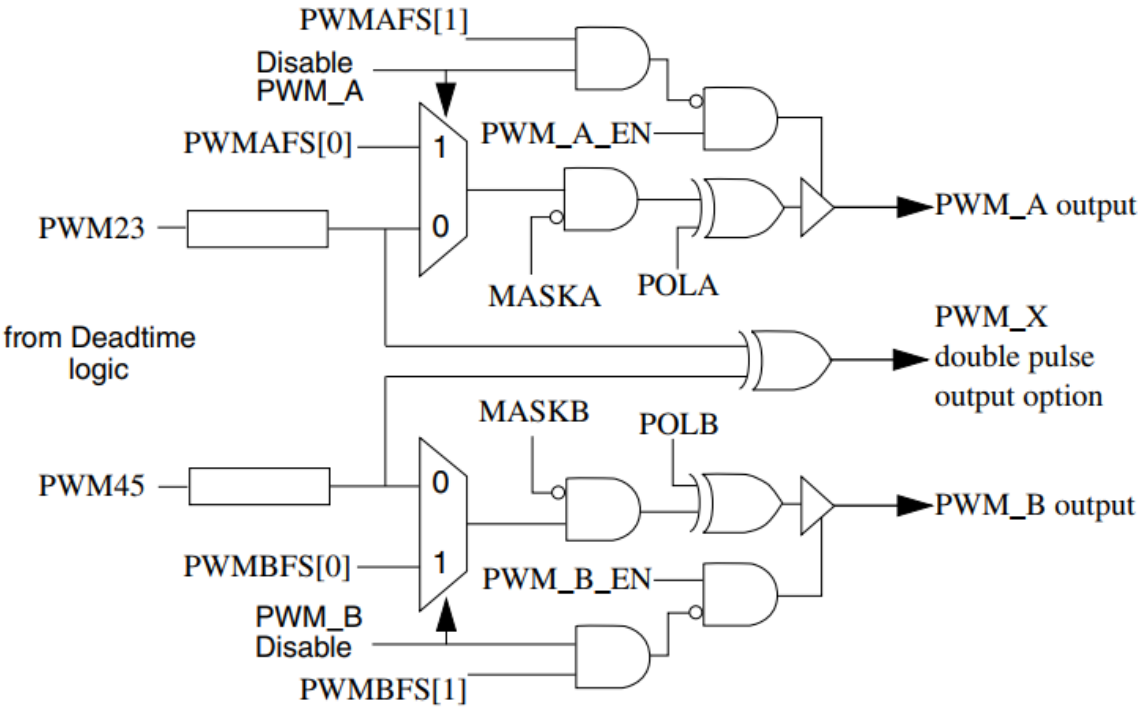
如果你的子模块没有NanoEdge的话,也可以用PWM抖动来模拟,编程来说差不多(实际上通过设置 FRCTRL[FRAC1_EN],FRCTRL[FRAC23_EN]和FRCTRL[FRAC45_EN]使能,而FRCTRL[FRAC_PU]是没有必要设置的,时钟频率也没那么多限制.),但是实际周期上不一样,比如要发生12.25的PWM,最后结果是13,12,12,12来模拟.

输出逻辑

下图显示了每个子模块的输出逻辑,反正有很多可以选择的位就是了,比如下面看到的PWM_A_EN什么的,与门嘛,短路输出的好帮手(只有与门一个输入为0输出一定为0).

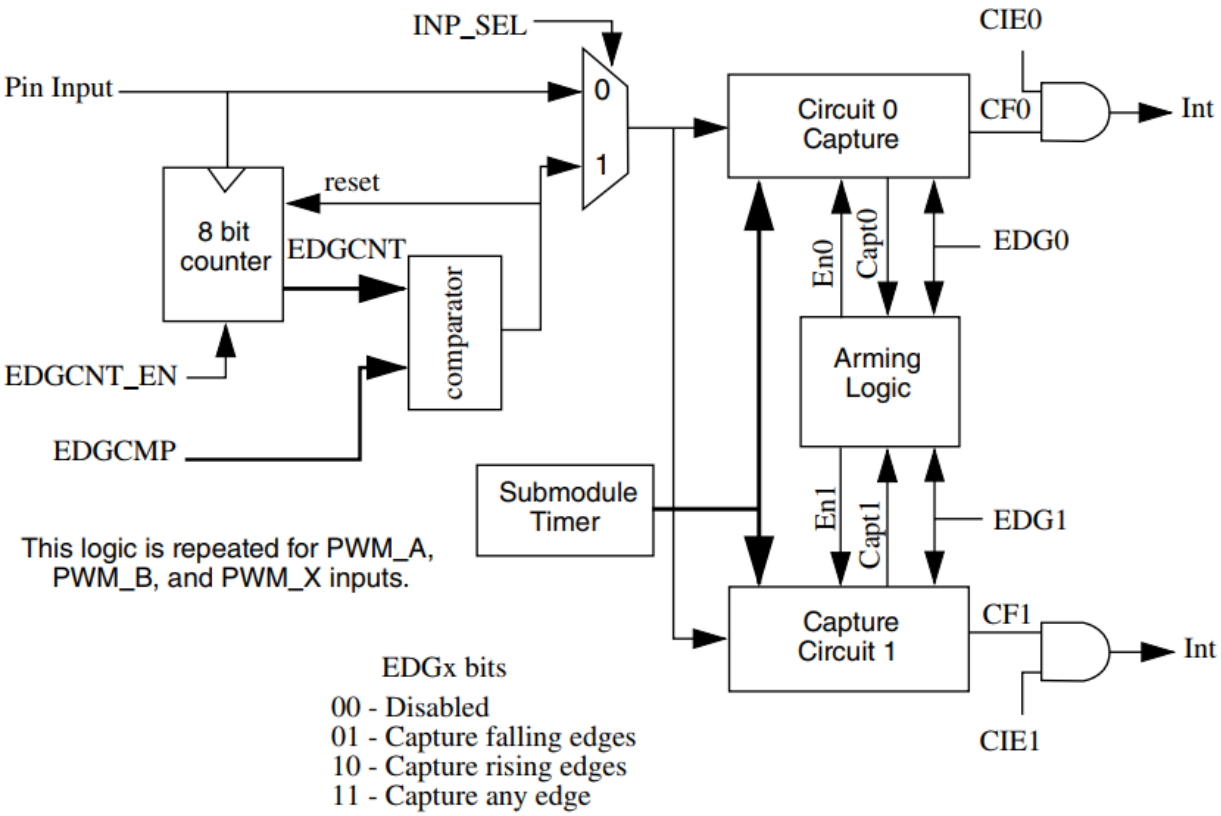
按照正常来说,PWM23输出PWM_A,PWM45输出PWM_B,在下面图中,PWM23和PWM45已经是死区逻辑处理后的结果,但是在真正输出的时候,有一个POLA/POLB的控制,他通过一个XOR门(不同则输出1,同则输出0)控制.

比如要输出高,到达XOR,POLA为高,所以输出低(所以是反了),要输出低,因POLA是高,所以输出高(对,就是反了),而如果POLA是低,则逻辑就对了,输出高就是高,低就是低.



增强捕获

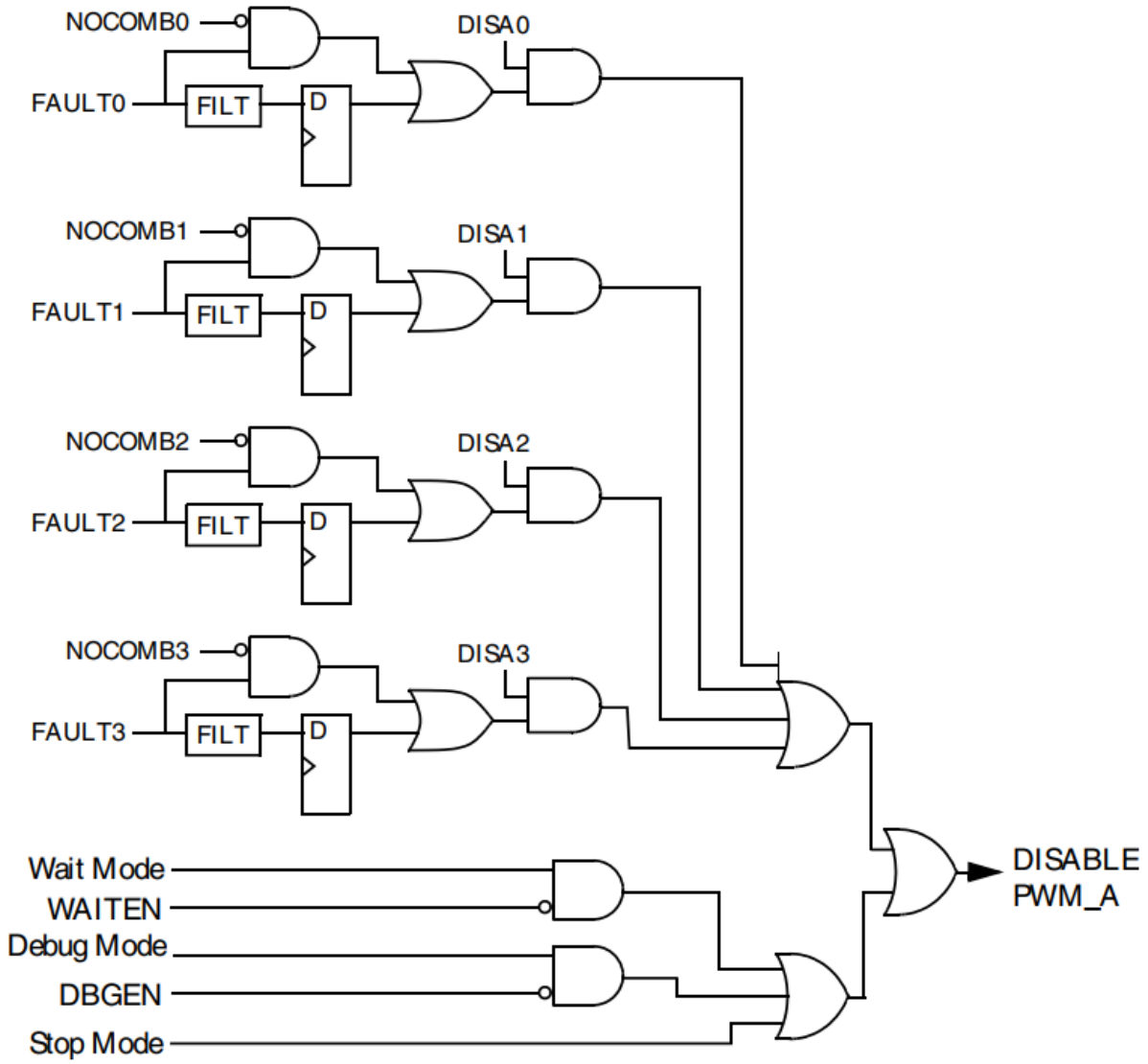
捕获就是通过PWM引脚捕获信号,然后记录数值,引脚输入后,分两路走,一个直接进入复用输入,软件可以选择将信号直接传递给捕获逻辑进行处理(就是后面的电路啦~),另外一个路径进入计数器(可以捕上升下降),这个计数器可以用用户指定的EDGECMPx比较,如果两个数值相等,就会输出一个事件,因为计数器是8位的,所以可以累计255次,然后输给后面的处理电路(这样再高速的跳变也没关系啦,反正N次后才触发)



上面还看到EDGx可以选择捕获的边沿,后面还有个PING-PONG缓冲区,如果自由运行时候,就会010101缓冲区交替储存了,这样等于两个寄存器存东西了.

故障保护

FAULT信号可以从引脚上控制,极性由FCTRL[FLVL]决定,当FAULT有效时候,PWM计数器不会停下来,但是引脚会被设置为OCTRL[PWMXFS]规定的值(高/低/三态),另外就算PWM模块没用,故障保护也会启动,所以,如果PWM发生中断,应该检查下是不是故障被锁存了,就是说,如果FAULT发生,那么故障标志就会有数值,写FSTS[FFLAX]清除FLAX[FLAX]~



PWM Pin	Controlling Register Bits
PWM_A	DISMAP0[DIS0A] and DISMAP1[DIS1A]
PWM_B	DISMAP0[DIS0B] and DISMAP1[DIS1B]
PWM_X	DISMAP0[DIS0X] and DISMAP1[DIS1X]

故障引脚还有个可编程的滤波器,FFILT[FILT_PER]设置多少,就滤过多少个周期,只要FAULT信号持续时间大于这个规定,才算有效啊,当然,可以设置这个寄存器为0,禁用滤波器.

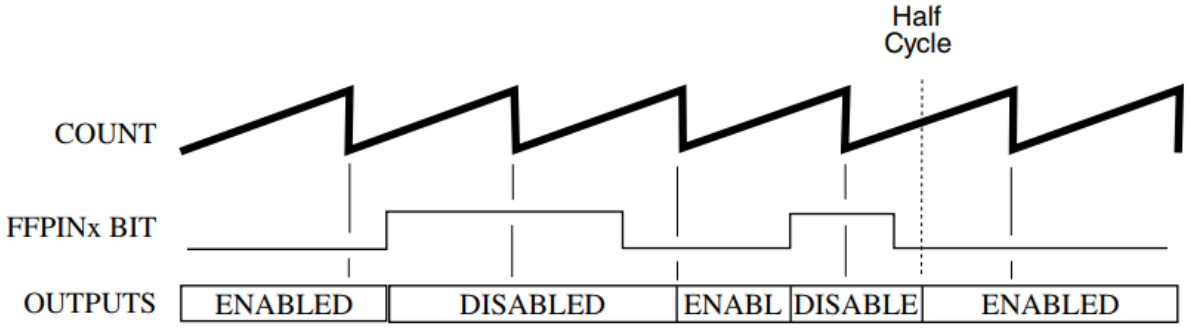
每个故障引脚都有一个可绕过的可编程滤波器。 滤波器的采样周期可以使用 FFILT [FILT_PER] 进行调整。 可以使用 FFILT [FILT_CNT] 调整在识别输入转换之前必须同意的连续采样数。 将 FFILT [FILT_PER] 设置为全部 0 将禁用给定 FAultX 引脚的输入滤波器。

如果设置了FEx,FAULTx引脚中断使能位,FSTS[FLAGx]将产生中断请求,中断请求会一直保持,直到下面的操作发生:

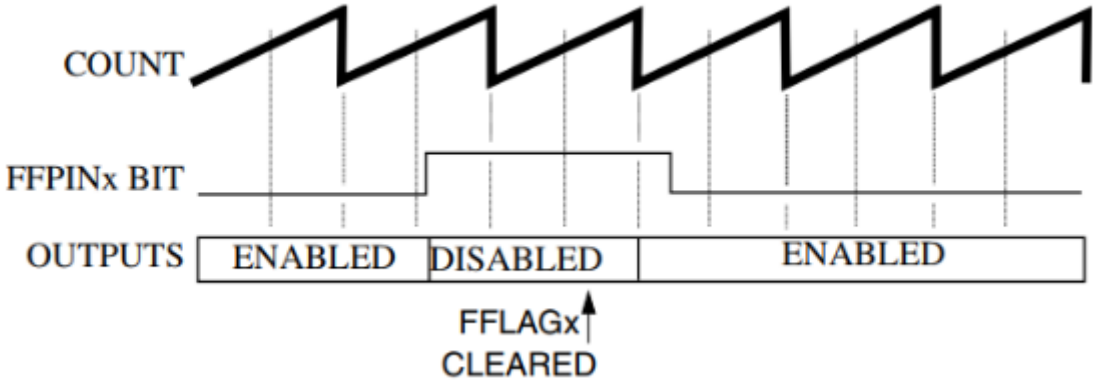
- FSTS[FLAGx] 清除
- FIEX 清除
- 重置

如果设置了FCTRL[FAUTOx],则故障可以自动清除,下面的图说明自动清除逻辑,分别几个情况:

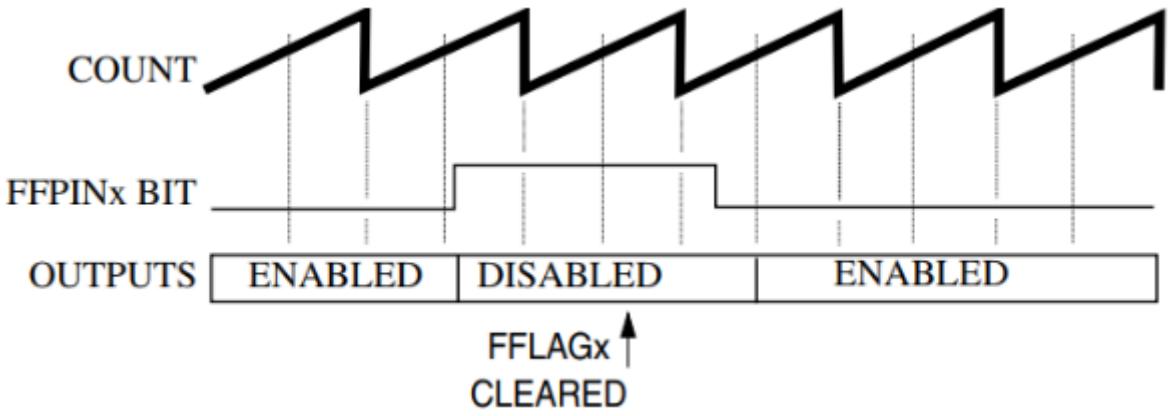
- 当故障位清除之后,下一次PWM完整周期来的时候,PWM重新输出,如果设置了FSTS[FULLx]则不会执行这个条件.
- 当故障位清除之后,遇到PWM的半周期来的时候,PWM重新输出,如果设置了FSTS[FHALFx]则不会执行这个条件.
- 当设置了FCTRL[FAUTOx]的时候,清除FSTS[FLAGx]没有影响.



如果没设置FCTRL[FAUTOx],就是手动清除,手动清除两个模式,分别是有没有设定FCTRL[FSAFEx]位,这个位区别主要是对FFPINxBIT的处理上,如果设置了,则不能无视FFPINxBIT的设定,只有FFPINxBIT同时释放(变低),才能恢复输出.如果没设置,就可以无视,至于能否在后续PWM半周期/全周期内重启,和前面自动清除说的一样,即受FSTS[FULLx]和FSTS[FHALHx]控制



.



当输出逻辑里面的SEL23(SEL45同理,不啰嗦了)字段选择OUT23或者PWM_EXTA时(反正不是从硬件输出的逻辑就行,因为另外两个选择都是从硬件输出的逻辑),可以软件控制故障保护,当PWM发生器参与的时候,故障清除依然会发生在PWM的半周期边界,MCTRL[RUN]等于1,但OUTX位可以在PWM发生器关闭时控制PWM引脚,MCTRL[RUN]等于0,因此,当PWM发生器关闭后直到重启时,发生故障清除.

FSTT[FTEST]可以模拟故障状态,用于测试.

PWM 发生器装载

MCTRL[LDOK]生效(设置这个位是为了同步)后可以加载以下PWM发生器参数:

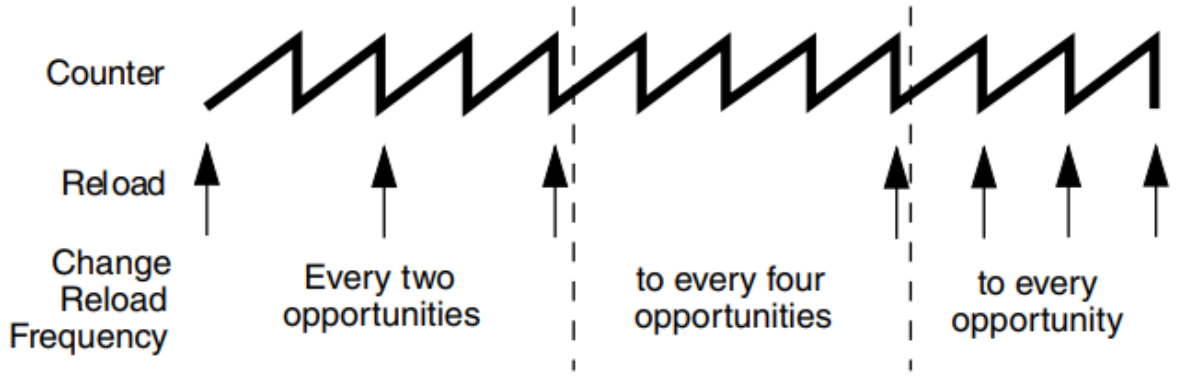
- 预分配器设置,即寄存器CTRL[PRSC]
- PWM周期和脉冲宽度,即寄存器INIT和VALx

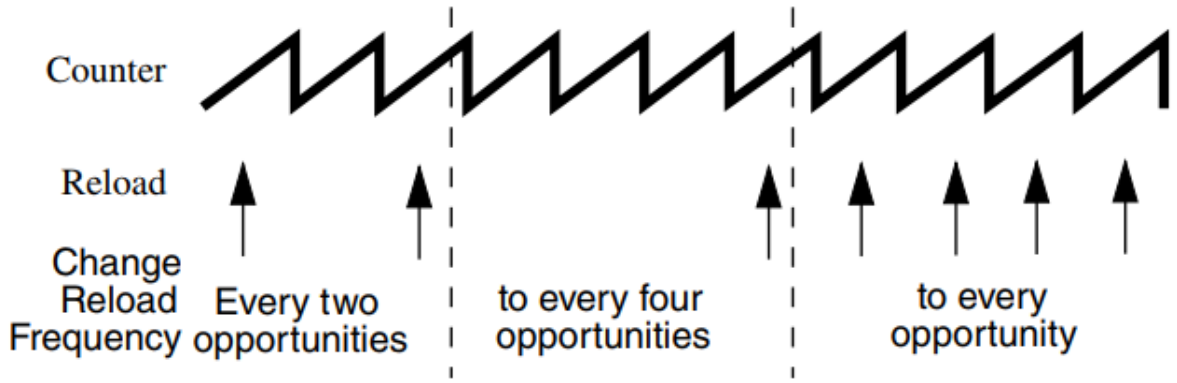
默认情况下,会在下一个PWM重载周期开始时传输到内部寄存器来更新设置,但是如果设置了CTRL[LDMOD],则可以在设置MCTRL[LDOK]时立即传输到内部寄存器,不管怎样,装载后,MTRL[LDOK]将自动清除.

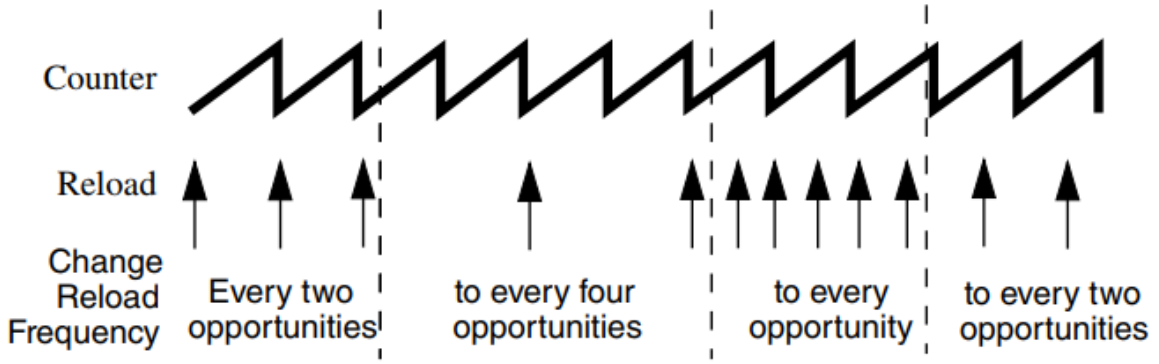
装载频率限制就像下面图说的一样,可以每次装载,也可以每N次(最多16)装载一下.这里受到CTRL[LDFQ]限制,这个装载是不会理会MCTRL[LDOK]的,另外是在半周期装载还是全周期装载,也有CTRL[FULL]和CTRL[HALF]控制.

分别解释FULL和HALF装载:

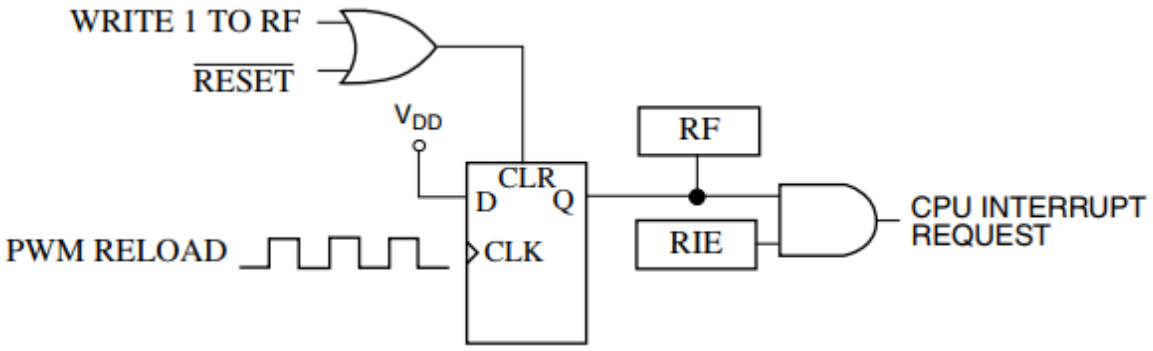
- 设置了FULL的时候,当计数等于VAL1时,每个PWM周期结束时都会出现重载机会(是机会,不是一定重新加载,什么时候加载受到加载频率也就是LDFQ限制)
- 设置了HALF的时候,当计数等于VAL0时,在版循环中出现重载机会(限制和上门说的一样,是机会,不是一定)
- 同时设置了HALF和FULL,就是上面两个条件的组合.







只要有重载机会就会设置RF标志,RIE标志会让RF标志设置时候发生中断.(只要有机会就行,不一定要真正发生重载)



当VALx/FRACVALx/CTRL[PRSC]寄存器中的任意一个更新时,STS[RUF]标志被设置为指示数据不一致,STS[RUF]将在重载成功时候清除,同时设置MCTRL[LDOK].如果设置了STS[RUF],并且在重新加载信号发生时MCTRL[LDOK]清除,就会发生了重载错误,并设置了STS[REF]用来标记错误,如果STS[RUF]在重载信号声明时清除,说明数据连贯,不会发生错误.

当MTRL[RUN]被清除时:

- STS[RF]标志/CPU中断不会清除
- 故障电路依然有效
- 软件/外部输出依然有效
- 死区控制依然有效

复位

复位会重置所有寄存器为默认值,引脚为三态.

在设置MCTRL[RUN]之前,需要初始化所有寄存器然后设置MCTRL[LDOK],当然,即使没有设置MCTRL[LDOK],设置MCTRL[RUN]之后也会设置STS[RF]标志,如果不想发生中断,再MCTRL[RUN]之前记得清除INTEN[RIE]

中断

中断标志位在SMxSTS有标记,使能位在SMxINTEN有标记,各中断名称在寄存器中有完整的描述.

DMA

每个子模块都可以请求DMA,由SMxDMAEN控制读写请求,具体方向和细节在寄存器中也有描述.

寄存器

eFlexPWM寄存器非常多,长度达到196h,所幸有些寄存器是功能类似的,所以归纳起来没那么吃力,不然吐血都整理不完了,具体还是对着手册看比较好.

寄存器概述:

- SM0CNT - SM3CNT
    - CNT[15:0] 当前计数值
- SM0INIT - SM3INIT
    - INIT[15:0] 初始值,写入后会在合适时机加载.

- SM0VALn - SM3VALn (n = 0~5)
    - VALn[15:0] 用来比较的值
- SM0FRACVALn - SM3FRACVALn (n = 1~5)
    - FRACVAn[15:11] 分数值,更精细化控制.(NanoEdge相关)
- SM0FRCTRL - SM3FRCTRL
    - TEST[15] 工厂测试,对于我们没有用.
    - FRAC_PU[8] 1 = 使能分数延迟功能(总开关)
    - FRAC45_EN[4] 针对PWM45的分开关,1 = 使能
    - FRAC23_EN[2] 同上(这里针对PWM23)
    - FRAC1_EN[1] 同上(这里针对PWM1)
- SM0OCTRL - SM3OCTRL
    - PWMA_IN[15] 当然PWMA输入电平
    - PWMB_IN[14] 当前PWMB输入电平
    - PWMX_IN[13] 当前PWMX输入电平
    - POLA[10] 0 = PWMA 输出不取反 1 = PWMA 输出取反
    - POLB[9] 同上(这里针对PWMB)
    - POLX[8] 同上(这里针对PWMX)
    - PWMAFS[5:4] PWMA Fault时候状态: 0 = 强制低 1 = 强制高 10/11 = 强制三态
    - PWMBFS[3:2] 同上(这里针对PWMB)
    - PWMXFS[1:0] 同上(这里针对PWMX)
- SM0STS - SM3STS
    - RUF[14] 寄存器更新标志位 (INIT,VALx,FRACVALx,CTRL[PRSC]更新都会产生这个标志,当MCTRL[LDOK]把这些缓冲的寄存器加载后可以清除)
    - REF[13] 加载错误,1 = 有错误,写1清除.
    - RF[12] 重载标志,1 = 自上次RF清除后又发生重载,写1清除.
    - CFA1[11] A1捕获,捕获发生时置位,写0清除.
    - CFA0[10] A0捕获,捕获发生时置位,写0清除.
    - CFB1[9] B1捕获,捕获发生时置位,写0清除.
    - CFB0[8] B0捕获,捕获发生时置位,写0清除.

- CFX1[7] X1捕获,捕获发生时置位,写0清除.
- CFX0[6] A0捕获,捕获发生时置位,写0清除.
- CMPF[5:0] 发生比较事件,写1清除,对应BIT置位 = 比较事件发生.

- SM0INTEN - SM3INTEN
  - 和 SM0STS - SM3STS 相对应,每一个STS都有一个INTEN,用来使能对应中断.
- SM0CTRL - SM3CTRL
  - LDFQ[15:12] 每X个PWM周期从寄存器影子中复制到真实寄存器(X=0~15,填0即每个周期搬运,填15即每16周期搬运)
  - HALF[11] 半周期加载使用(1 = 使能)
  - FULL[10] 全周期加载使用(1 = 使能)
  - DT[9:8] 死区时间(只读)监测值,采样发生在DT[0]/DT[1]结束的时候.
  - COMPMODE[7]
    - 0 = 当PWM等于比较寄存器的VAL时,产生PWM边沿.(意味着,周期结束的时候,PWM的输出电平会保持,直到下一次匹配)
    - 1 = 当PWM等于或者大于比较寄存器VAL时,产生PWM边沿.(意味着,如果起始计数器的值大于(不一定等于)比较值,则周期结束时候可以从高变低)
  - PRSC[6:4] PWM时钟分频系数(不分频,寄存器分频系数等于2^(n-1),最高128分频.)
  - SPLIT[3] 关联下面的DBLEN[0],只有DBLEN[0]为1时候有效,具体双开关PWM有介绍.
  - LDMOD[2] 决定缓冲寄存器加载时机,在MCTRL[LDOK]置位后 0 = 下一个周期再加载 1 = 尽快加载
  - DBLX[1] 针对PWMX的DBLEN,用途和DBLEN一样,只不过是针对PWMX的.
  - DBLEN[0] 双开关使能位

- SM0CTRL2 - SM3CTRL2
    - DBGEN[15] 0 = 调试时计数器不跑 1 = 调试时计数器跑
    - WAITEN[14] 0 = 等待模式时计数器不跑 1 = 等待模式时计数器跑
    - INDEP[13] 0 = PWM成对操作 1 = PWM独立操作 (针对PWMA/PWMB的,PWMX总是独立)
    - PWM23_INIT[12] 初始状态电平
    - PWM45_INIT[11] 同上(这里针对PWM45)
    - PWMX_INIT[10] 同上(这里针对PWMX)
    - INIT_SEL[9:8] 计数器的重置信号选择(参考前面的图)
    - FRCEN[7] 0 = 不可以强制重载 1 = 可以强制重载 (如果MCTRL[LDOK]有设置,那么还会发生本地重载)
    - FORCE[6] FORCE_SEL为000的时候,写1就会产生FORCE_OUT信号,然后产生下面两个事件:
        - PWMA/B的输出将由DTSRCSEL[SMxSEL23]和DTSRCSEL[SMxSEL45]决定.
        - 如果CTRL2[FRCEN]置位,计数器数值变为INIT寄存器的数值.
    - FORCE_SEL[5:3] ref:[强制输出逻辑]
    - RELOAD_SEL[2] ref:[寄存器重载逻辑]
    - CLK_SEL[1:0] 时钟源选择
        - 00 IPBus 时钟
        - 01 EXT_CLK
        - 10 子模块0
- SM0TCTRL - SM3TCTRL
    - PWAOT0[15] 输出触发源选择
        - 0 = PWM_OUT_TRIG0 连接 PWM_OUT_TRIG0
        - 1 = PWMA 连接 PWM_OUT_TRIG0

    - PWBOT1[14] 同上,不过这里换成TRIG1和PWMB
    - TRGFRQ[12] 触发频率,要求CTRL[LDFQ] != 0 (等于0就是每个周期都会重载缓存的寄存器.)
        - 无视CTRL[LDFQ],依然每个周期触发.
        - 关心CTRL[LDFQ],在每个周期的最后一次才触发.
    - OUT_TRIG_EN[5:0]　对应BIT置位 = 输出触发使能 (PWM_OUT_TRIGx由6个不同引脚组成.)
- SM0DISMAPn - SM3DISMAPn (n = 0~1)
    - DISnX[11:8] 屏蔽PWM_X的FAULT事件,总共有4个这样的寄存器,对应4个FAULTx引脚.
    - DISnB[7:4] 同上,不过这里说的是PWM_B
    - DISnA[3:0] 同上,不过这里说的是PWM_A
- SM0DTCNTn - SM3DTCNTn (n = 0~1)
    - DTCNTn[15:0]
        - 无小数延迟情况,仅DTCNTn[10:0]有效,最大值0x7FFF,代表插入2047个死区周期.
        - 如果使能了小数延迟,那么最大是2047 + (31/32),推算最大是0xFFFF
        - n = 0对应PWM23 / n = 1对应PWM45
- SM0CAPTCTRLn - SM3CAPTCTRLn (n = A,B,X) (m = 0,1)
    - Cn1CNT[15:13] 指示捕获寄存器1的FIFO里面存了多少个东西.
    - Cn0CNT[12:10] 指示捕获寄存器0的FIFO里面存了多少个东西.
    - CFnWM[9:8] STS的设置阈值,只有FIFO深度大于这里的设定,STS对应标志才会设置.
    - EDGCNTn_EN[7] 边缘计数器使能(N边缘后捕获,上面的图有介绍过)

    - INP_SELn[6] 0 = 选择PWM_n作为信号源 1 = 边沿计数器/比较输出作为信号源,当选择这一个功能时候,会自动启动内部边缘计数器,并且忽略由CAPTCTRLn[EDGnm]字段指定的上升沿和/或下降沿,但是为了启用一个或两个捕获寄存器,软件仍必须在CAPTCTLRn[EDGnm]放置一个非0的值.(因为0就是禁止了啊,就是下面说的两个位.)
    - EDGn1[5:4] 0 = 禁止 1 = 捕获下降沿 2 = 捕获上升沿 3 = 捕获任何边沿
    - EDGn0[3:2] 同上,不过上面是边缘1,现在是是边缘0.
    - ONESHOTn[1] 0 = 一直捕获 1 = 只捕一下
    - ARMn[0] 输入捕获逻辑使能(这个位竟然不带EN结尾),0 = 禁止 1 = 使能(使能后还要EDGnm再细分使能)
- SM0CAPTCOMPn - SM3CAPTCOMPn (n = A,B,X)
    - EDGCNTn[15:8] 当前边沿计数
    - EDGCMPn[7:0] 边缘比较值
- SM0CVALn - SM3CVALn (n = 0~5) (m = A,B,X) (x = 0,1)
    - CAPTVALn[15:0] 该只读寄存器存储从子模块计数器捕获的值,确切的发生时间是由CAPTCTRLm[EDGmx]定义的,每次捕获将使CAPTCTRLm[CnxCNT]的值加1,直到达到最大值,每次读取该寄存器将使CAPTCTRLm[CnxCNT]减1,直到达到0.
        - mx的值:n = 0 => X0 / n = 1 => X1 / n = 2 => A0 / ... /n = 5 => B1
- SM0CVALnCYC - SM3CVALnCYC (n = 0~5)
    - CVALnCYC[3:0] 存储与CVALn中捕获的值相对应的周期号,在每次PWM周期结束的时候,这个数值都会递增.
- SMnDMAEN (n = 0~3)
    - VALDE[9] 0 = 禁止DMA所有写操作 1 = DMA可以对VALn和FRACVALn进行写操作.
    - FAND[8] 当CAPTDE选择是1的时候,这个位的值才有意义.

        - 0 = 全部阈值一起进行OR运算输出结果,有一个过全部过.
        - 1 = 全部阈值一起进行AND运算输出结果,全过才过.
    - CAPTDE[7:6] 下面全部位的总开关.
        - 0 = 禁用
        - 1 = 受FIFO阈值控制的传输,到达预定条件会发起DMA传送
        - 2 = 本地同步(VAL1比较事件)会发起DMA传送
        - 3 = 本地重载会发起DMA传送
    - CA1DE[5] 如果使能,SMnCVAL3的数值可以被DMA传送.
    - CA0DE[4] 如果使能,SMnCVAL2的数值可以被DMA传送.
    - CB1DE[3] 如果使能,SMnCVAL5的数值可以被DMA传送.
    - CB0DE[2] 如果使能,SMnCVAL4的数值可以被DMA传送.
    - CX1DE[1] 如果使能,SMnCVAL1的数值可以被DMA传送.
    - CX0DE[0] 如果使能,SMnCVAL0的数值可以被DMA传送.
        - 特别注意:0/1 对应X通道,2,3对应A通道,4,5对应B通道.
        - SMn 意思是子模块n.
        - DMA传输和中断是能不能同时开启.
- SM0PHASEDLY - SM3PHASEDLY
    - PHASEDLY[15:0] 相位延迟
- OUTEN
    - PWMA_EN[11:8] 1 = PWMA输出使能 对应的BIT分别是不同的子模块.
    - PWMB_EN[7:0] 1= PWMB输出使能 对应的BIT分别是不同的子模块.
    - PWMX_EN[3:0] 0 = PWMX输出使能 对应的BIT分别是不同的子模块.

- MASK
  - UPDATE_MASK[15:12]
    - 0 = 正常操作,在子模块发生FORCE_OUT事件之前,不会更新子模块对应的MASK位.
    - 对应BIT置位 = 相应子模块的MASK位将在下一个时钟沿更新.
    - MASK位功能是输出功能中描述的,用以输入与门(MASK输入前置非门)以短路某个逻辑.置1则输出0,则输出会被短路.
  - MASKA[11:8] PWMA的MASK位,每个BIT对应都是每一个子模块.
  - MASKB[7:4] 同上,不过这里指的是PWMB.
  - MASKX[3:0] 同上,不过这里指的是PWMA.
- SWCOUT
  - SM3OUT23[7] 低信号(如果这个BIT设置1就是高信号)提供给子模块3的死区发生器.
  - SM3OUT45[6] 低信号(如果这个BIT设置1就是高信号)提供给子模块3的死区发生器.
  - SM2OUT23[5] 低信号(如果这个BIT设置1就是高信号)提供给子模块2的死区发生器.
  - SM2OUT45[4] 低信号(如果这个BIT设置1就是高信号)提供给子模块2的死区发生器.
  - SM1OUT23[3] 低信号(如果这个BIT设置1就是高信号)提供给子模块1的死区发生器.
  - SM1OUT45[2] 低信号(如果这个BIT设置1就是高信号)提供给子模块1的死区发生器.
  - SM0OUT23[1] 低信号(如果这个BIT设置1就是高信号)提供给子模块0的死区发生器.
  - SM0OUT45[0] 低信号(如果这个BIT设置1就是高信号)提供给子模块0的死区发生器.

        - 上面描述的这些寄存器发生的信号,都是提供给DTSRCSEL使用的.
- DTSRCSEL
    - SM3SEL23[15:14]
        - 0 = 死区逻辑使用生成的SM3PWM23信号.
        - 1 = 死区逻辑使用生成的SM3PWM23信号取反.
        - 2 = 死区逻辑使用SWCOUT[SM3OUT23]提供的信号.
        - 3 = 死区逻辑使用PWM3_EXTA信号.
    - SM3SEL45[13:12] 与上面的类似.
    - SM2SEL23[11:10] 与上面的类似.
    - SM2SEL45[9:8] 与上面的类似.
    - SM1SEL23[7:6] 与上面的类似.
    - SM1SEL45[5:4] 与上面的类似.
    - SM0SEL23[3:2] 与上面的类似.
    - SM0SEL45[1:0] 与上面的类似.
- MCTRL
    - IPOL[15:12] 针对每个子模块单独设置,每一个BIT对应每一个子模块,如果取0,则PWM23用于在相应的子模块中生成互补的PWM对,否则用PWM45来生成,互补的话,只需要控制其中一个,另一个是互补生成的.
    - RUN[11:8] 哪个BIT置为,哪个子模块的PWM生成器就可以跑.
    - CLDOK[7:4] 如果MCTRL[LDOK]不是自动清除的话,写1可以清除,如果是自动清除的话,这个位永远读回0,依然是每个BIT对应每个子模块.
    - LDOK[3:0] 0 = 不加载新的数值 1 = 从缓冲中加载新的数值. 每次成功重新加载后,都会自动清除,不想自动清除的方法就是在加载前,用CLDOK来清除,这样也不会从从缓冲中加载新的数值到目标.
- MCTRL2

- MONPLL[1:0]
  - 0 = 未锁定,不监视PLL,任何故障都不可感知.
  - 1 = 未锁定,监视PLL操作,如果PLL有问题,自动禁用小数模块.
  - 2 = 锁定,功能和0相同,但是锁定后位就不能改了.
  - 3 = 锁定,功能和1相同,但是锁定后位就不能改了.

- FCTRL0
  - FLVL[15:12] FAULT引脚输入有效电平,每个BIT都对应每个FAULT输入,如果这里置1,则输入高到FAULT引脚才算FAULT有效.
  - FAUTO[11:8] FAULT自动清除逻辑,这个在前面故障保护里面说得明白,也是针对每个子模块.
  - FSAFE[7:4] 安全模式,如果是0,则是常规模式,PWM输出不受FAULT影响,如果是1,则会禁用输出.
  - FIE[3:0] FAULT发起中断,每个BIT对应每个子模块.
- FSTS0
  - FHALF[15:12] 0 = 不可以在半周期时候故障恢复 1 = 可以在半周期时候故障恢复
  - FFPIN[11:8] 故障PIN状态(已经滤波)
  - FFULL[7:4] 0 = 不可以在全周期时候故障恢复 1 = 可以在全周期时候故障恢复
  - FFLAG[3:0] 0 = FAULT没有任何FAULT引脚的事件 对应BIT置位 = 这些PIN发生了FAULT事件
- FFILT0
  - GSTR[15] 0 = 不使用毛刺过滤 1 = 启用过滤,故障输入长度至少2个(或者更多)IPBus时钟周期才算真输入
  - FILT_CNT[10:8] 过滤深度,设置范围0 - 7,代表3 - 10个IPBus时钟周期.

- FILT_PER[7:0] 输入滤波器采样周期,如果值是0代表禁用,每个输入采样多少次由这里确定.

- FTST0
  - FTEST[0] 写1人工发生FAULT事件,写0清除.
- FCTRL20
  - NOCOMB[3:0]
    - 0 = 从故障输入到PWM输出存在组合路径,故障输入与经过滤波和锁存的故障信号结合才能禁用PWM输出.
    - 1 = 从故障输入到PWM输出不存在组合路径,故障输入与经过滤波和锁存的故障信号都能禁用PWM输出.

示例实现

- 特别注意:这个示例只是其中一个简单得Example,是官方板上实现的Example,由于这个定时器非常复杂,不是三言两语说的完的,所以更多是需要实践.
- 要测量信号只能从电阻R793,R794,R796,R798一侧测试,如果要连出来,需要改板子.
- 涉及的技术:
  - FAULT 输入
  - 死区设置

```
/*
 * Copyright (c) 2015, Freescale Semiconductor, Inc.
 * Copyright 2016-2017 NXP
 * All rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include "fsl_debug_console.h"
#include "board.h"
#include "fsl_pwm.h"

#include "pin_mux.h"
```

```c
#include "fsl_xbara.h"
/*******************************************************************************
 * Definitions
 ******************************************************************************/
/* The PWM base address */
#define BOARD_PWM_BASEADDR PWM1

#define PWM_SRC_CLK_FREQ CLOCK_GetFreq(kCLOCK_IpgClk)

/*******************************************************************************
 * Prototypes
 ******************************************************************************/

/*******************************************************************************
 * Variables
 ******************************************************************************/

/*******************************************************************************
 * Code
 ******************************************************************************/
/* 这是关于3相PWM的关键逻辑. */
static void PWM_DRV_Init3PhPwm(void)
{
  uint16_t deadTimeVal;
  pwm_signal_param_t pwmSignal[2];
  uint32_t pwmSourceClockInHz;
  uint32_t pwmFrequencyInHz = 1000;

  pwmSourceClockInHz = PWM_SRC_CLK_FREQ;

  /* 设置 650ns 死区. */
  deadTimeVal = ((uint64_t)pwmSourceClockInHz * 650) / 1000000000;

  pwmSignal[0].pwmChannel = kPWM_PwmA;
  pwmSignal[0].level = kPWM_HighTrue;
```

```
  pwmSignal[0].dutyCyclePercent = 50; /* 1 percent dutycycle */
  pwmSignal[0].deadtimeValue = deadTimeVal;

  pwmSignal[1].pwmChannel = kPWM_PwmB;
  pwmSignal[1].level = kPWM_HighTrue;
  /* 这里设置多少都没用,因为他们是成对的,所以还是受PWMA的控制. */
  pwmSignal[1].dutyCyclePercent = 50;
  pwmSignal[1].deadtimeValue = deadTimeVal;

  /* 子模块0发生A相,2个输出 */
  PWM_SetupPwm(BOARD_PWM_BASEADDR, kPWM_Module_0, pwmSignal, 2, kPWM_Signed
  pwmSourceClockInHz);

  /* 子模块1发生B相,2个输出 */
  PWM_SetupPwm(BOARD_PWM_BASEADDR, kPWM_Module_1, pwmSignal, 1, kPWM_Signed
  pwmSourceClockInHz);

  /* 子模块2发生C相,2个输出 */
  PWM_SetupPwm(BOARD_PWM_BASEADDR, kPWM_Module_2, pwmSignal, 1, kPWM_Signed
  pwmSourceClockInHz);
 }

/*!
 * @brief Main function
 */
int main(void)
{
  /* 预定义的PWM结构和变量 */
  pwm_config_t pwmConfig;
  static uint16_t delay;
  uint32_t pwmVal = 4;
  uint16_t i;

  /* 配置引脚,调试串口,PWM输出引脚等等. */
  BOARD_ConfigMPU();
  BOARD_InitPins();
  BOARD_BootClockRUN();
  BOARD_InitDebugConsole();
```

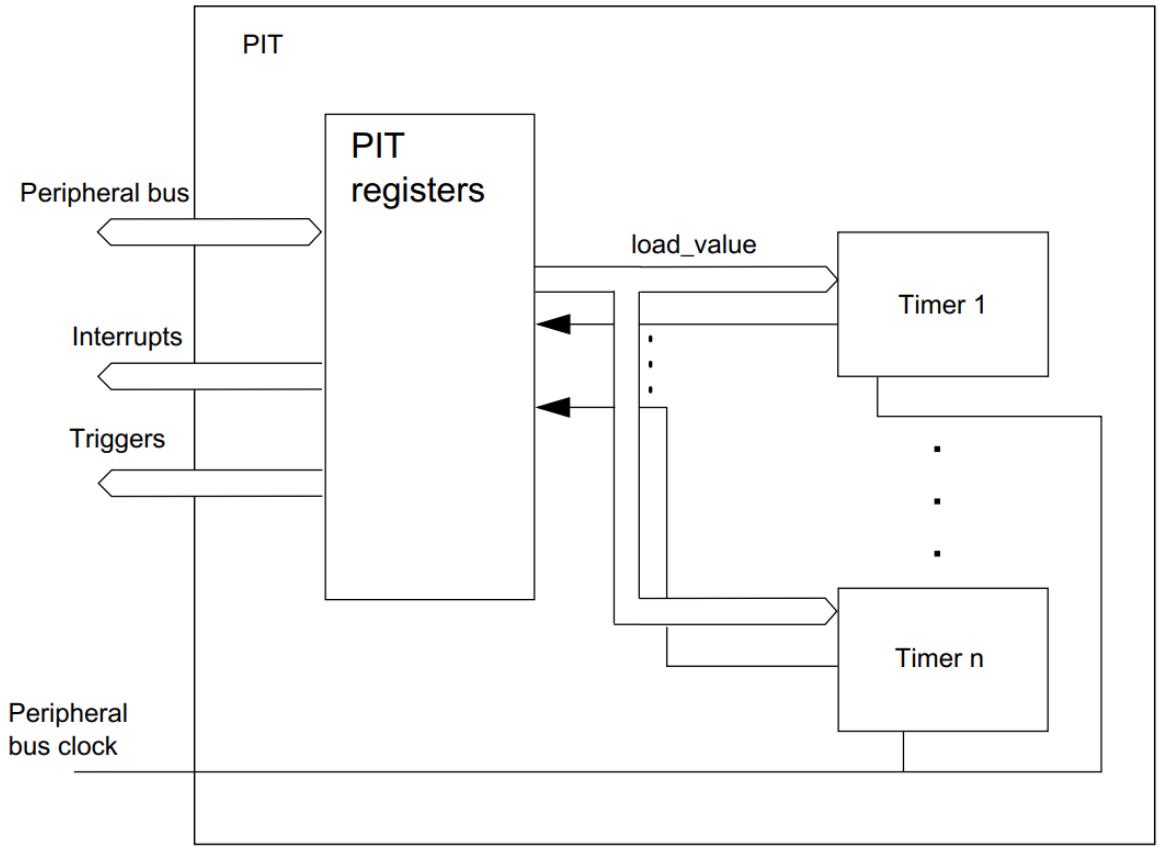
```
87
88  CLOCK_SetDiv(kCLOCK_AhbDiv, 0x2); /* Set AHB PODF to 2, divide by 3 */
89  CLOCK_SetDiv(kCLOCK_IpgDiv, 0x3); /* Set IPG PODF to 3, divede by 4 */
90
91  /* 用XBAR把FAULT路由出去,然后再设置FAULT的输入等,这个暂时不解释,涉及XBAR以后再说. */
92  XBARA_Init(XBARA);
93  XBARA_SetSignalsConnection(XBARA, kXBARA1_InputLogicHigh, kXBARA1_OutputFlexpwm1Fault0);
94  XBARA_SetSignalsConnection(XBARA, kXBARA1_InputLogicHigh, kXBARA1_OutputFlexpwm1Fault1);
95  XBARA_SetSignalsConnection(XBARA, kXBARA1_InputLogicHigh, kXBARA1_OutputFlexpwm1Fault2);
96  XBARA_SetSignalsConnection(XBARA, kXBARA1_InputLogicHigh, kXBARA1_OutputFlexpwm1Fault3);
97
98  PRINTF("FlexPWM driver example\n");
99
100  /*
101  * pwmConfig.enableDebugMode = false;
102  * pwmConfig.enableWait = false;
103  * pwmConfig.reloadSelect = kPWM_LocalReload;
104  * pwmConfig.faultFilterCount = 0;
105  * pwmConfig.faultFilterPeriod = 0;
106  * pwmConfig.clockSource = kPWM_BusClock;
107  * pwmConfig.prescale = kPWM_Prescale_Divide_1;
108  * pwmConfig.initializationControl = kPWM_Initialize_LocalSync;
109  * pwmConfig.forceTrigger = kPWM_Force_Local;
110  * pwmConfig.reloadFrequency = kPWM_LoadEveryOportunity;
111  * pwmConfig.reloadLogic = kPWM_ReloadImmediate;
112  * pwmConfig.pairOperation = kPWM_Independent;
113  *
114  * 上面给出的是默认配置的信息,我们可以在默认基础上修改.
115  */
116  PWM_GetDefaultConfig(&pwmConfig);
117
118  /* 全周期时候才可以重载(另外一个是半周期时候可以重载,默认是LDOK可以重载.) */
119  pwmConfig.reloadLogic = kPWM_ReloadPwmFullCycle;
120  /* 互补输出(成对存在) */
121  pwmConfig.pairOperation = kPWM_ComplementaryPwmA;
122  pwmConfig.enableDebugMode = true;
123
```

```
124   /* 子模块0用上面的配置初始化. */
125   if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_0, &pwmConfig) == kStatus_
Fail)
126   {
127   PRINTF("PWM initialization failed\n");
128   return 1;
129   }
130
131   /* 子模块1用另一个方式初始化,用子模块0作为周期重载时钟. */
132   pwmConfig.clockSource = kPWM_Submodule0Clock;
133   pwmConfig.initializationControl = kPWM_Initialize_MasterSync;
134   if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_1, &pwmConfig) == kStatus_
Fail)
135   {
136   PRINTF("PWM initialization failed\n");
137   return 1;
138   }
139
140   /* 子模块2和子模块1一样. */
141   if (PWM_Init(BOARD_PWM_BASEADDR, kPWM_Module_2, &pwmConfig) == kStatus_
Fail)
142   {
143   PRINTF("PWM initialization failed\n");
144   return 1;
145   }
146
147   /* 调用三相PWM演示其他初始化 */
148   PWM_DRV_Init3PhPwm();
149
150   /* 设置LDOK来强制全部重载. */
151   PWM_SetPwmLdok(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 | kPWM_Control
_Module_1 | kPWM_Control_Module_2, true);
152
153   /* 启动PWM */
154   PWM_StartTimer(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 | kPWM_Control
_Module_1 | kPWM_Control_Module_2);
155
156   delay = 0x0fffU;
157
158   while (1U)
159   {
160   /* 延迟一段时间递增比较数(改变占空比) */
```

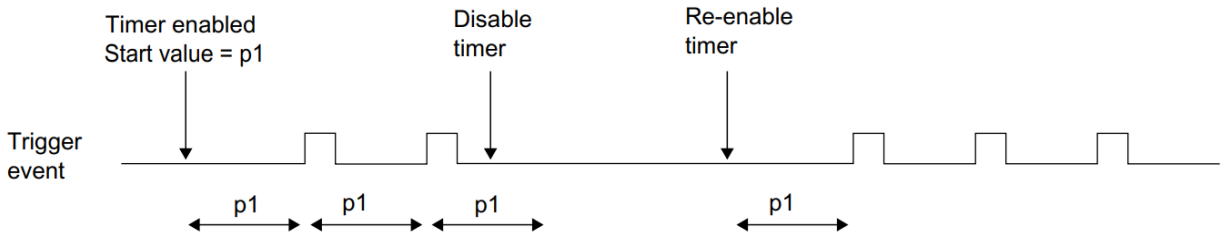
```
161   for (i = 0U; i < delay; i++)
162   {
163   __ASM volatile("nop");
164   }
165   pwmVal = pwmVal + 4;
166
167   /* 如果占空比足够大,就让他从最小重新开始. */
168   if (pwmVal > 100)
169   {
170   pwmVal = 4;
171   }
172
173   /* 把他们写道缓冲寄存器里面. */
174   PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR, kPWM_Module_0, kPWM_PwmA, kP
WM_SignedCenterAligned, pwmVal);
175   PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR, kPWM_Module_1, kPWM_PwmA, kP
WM_SignedCenterAligned, (pwmVal >> 1));
176   PWM_UpdatePwmDutycycle(BOARD_PWM_BASEADDR, kPWM_Module_2, kPWM_PwmA, kP
WM_SignedCenterAligned, (pwmVal >> 2));
177
178   /* 设置LDOK来让他们加载. */
179   PWM_SetPwmLdok(BOARD_PWM_BASEADDR, kPWM_Control_Module_0 | kPWM_Control
_Module_1 | kPWM_Control_Module_2, true);
180   }
181 }
```

Periodic Interrupt Timer (PIT)

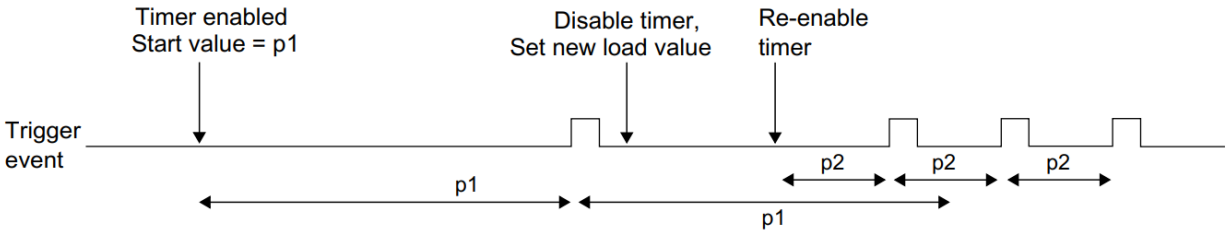
不是运行就是停止,算是最简单的定时器了,可以触发DMA,中断,每个定时器可以独立计算超时.



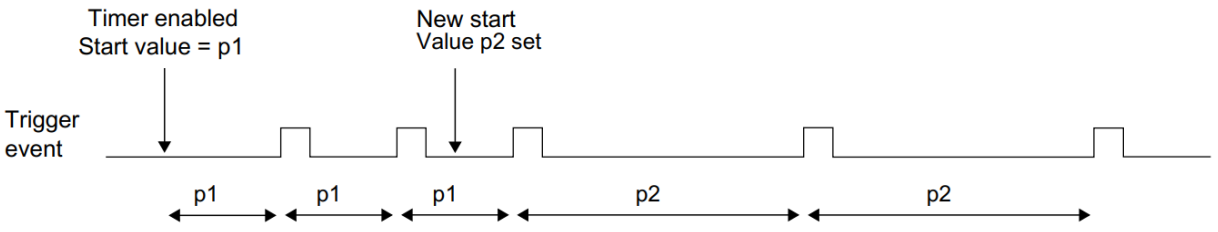
启动后,加载LDVAL数值,然后计数到0,然后发生中断,周而复始.TCTRLn[TIE]控制中断的发生.TFLGn[TIF]是中断标志.



在禁止的时候修改周期.



在不禁止情况下修改周期,会在当前周期结束后才应用新周期.



调试模式:MCR[FRZ] - 可以在断点时候暂停定时器

级联模式:如果N-1定时器计算到0,N定时器就会减1,两个级联在最大计数模式下,即使75MHz的时钟(默认配置的最大外设时钟),也要7793年才能计算完毕,如果要更大的级联明显没有意义(虽然可以这么做),四个级联地球毁灭都算不完.

示例实现

- 定时器基准:50MHz,计数器每一下是20ns.
- Timer 0 256000周期中断(5.12 ms)
- Timer 1 1500000周期不中断(30 ms)

```
volatile int pitIsrCnt = 0;

void PIT_IRQHandler(void)
{
    /* Clear interrupt flag.*/
    if (PIT->CHANNEL[0].TFLG)
    {
        PIT->CHANNEL[0].TFLG = 1;
        pitIsrCnt++;
     }
     __DSB();
}

static void MainTask(void *pvParameters)
{
    CLOCK_EnableClock(kCLOCK_Pit);

    PIT->MCR = 0x00;
    PIT->CHANNEL[0].LDVAL = 255999;
    PIT->CHANNEL[0].TCTRL = PIT_TCTRL_TIE(1);
    PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN(1);

    PIT->CHANNEL[1].LDVAL = 1499999;
    PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_TEN(1);

    NVIC_EnableIRQ(PIT_IRQn);

    for (;;)
```

```
    {
        vTaskDelay(pdMS_TO_TICKS(1000));
        GPIO1->DR_TOGGLE |= (1UL << 11U);
    }
}
```

- 定时器基准:100MHz.
- Timer0 和 Timer 1 都使能
- 时间计算:Timer0触发600000000下,Timer1就会触发,相乘等于一分钟.
- 中断频率:1分钟

```
volatile int pitIsrCnt = 0;

void PIT_IRQHandler(void)
{
    /* Clear interrupt flag.*/
    if (PIT->CHANNEL[1].TFLG)
    {
        PIT->CHANNEL[1].TFLG = 1;
        pitIsrCnt++;
     }
     __DSB();
}

static void MainTask(void *pvParameters)
{
    CLOCK_EnableClock(kCLOCK_Pit);

    PIT->MCR = 0x00;
    PIT->CHANNEL[1].LDVAL = 9;
    PIT->CHANNEL[1].TCTRL = PIT_TCTRL_TIE(1);
    PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_CHN(1);
    PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_TEN(1);

    PIT->CHANNEL[0].LDVAL = 599999999;
    PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN(1);

    NVIC_EnableIRQ(PIT_IRQn);

    for (;;)
```

```
    {
        vTaskDelay(pdMS_TO_TICKS(1000));
        GPIO1->DR_TOGGLE |= (1UL << 11U);
    }
}
```

- 实现需求:超长时间计时器,比如uptime计算.
- 实现方法:定时器超长级联.

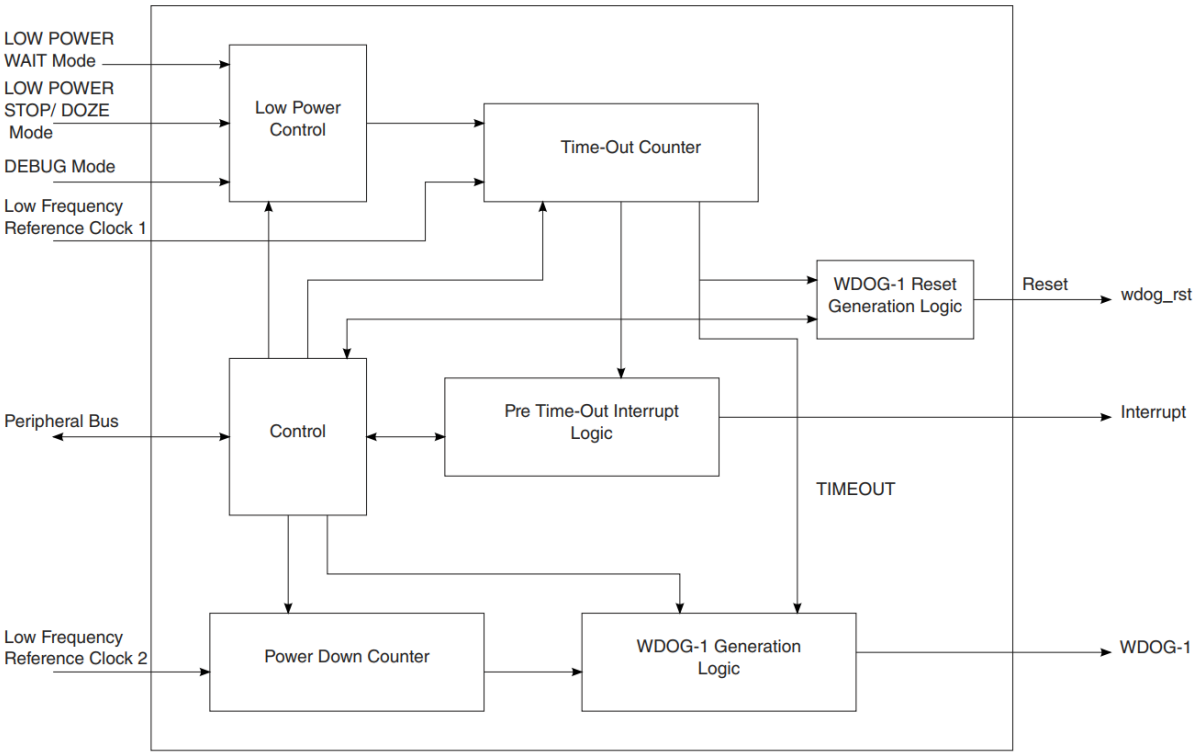
```
volatile uint64_t current_uptime = 0;

static void MainTask(void *pvParameters)
{
    CLOCK_EnableClock(kCLOCK_Pit);

    PIT->MCR = 0x00;
    PIT->CHANNEL[1].LDVAL = 0xFFFFFFFF;
    PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_CHN(1);
  PIT->CHANNEL[1].TCTRL |= PIT_TCTRL_TEN(1);

     PIT->CHANNEL[0].LDVAL = 0xFFFFFFFF;
     PIT->CHANNEL[0].TCTRL |= PIT_TCTRL_TEN(1);

     for (;;)
     {
         current_uptime = 0xFFFFFFFFFFFFFFFF - (((uint64_t)PIT->LTMR64H << 32) + PIT->LTMR64L);
         vTaskDelay(pdMS_TO_TICKS(1000));
     }
}
```

Watchdog Timer (WDOG1-2)

看门狗,功能简单,但是非常敬业,废话比较多,但是功能却不多.

- 可配置超时:0.5 - 128秒,超时一到就疯狗咬人(WDOG_RESET_B_DEB 复位)
- 时间精度:0.5秒.
- 低功耗模式:可以暂停/运行
- 调试模式:可以暂停/运行

- 咬人前提醒:可发生中断,可以配置 0 - 127.5秒(PS:相对于STM32固定的提前中断时间,这个明显更灵活)
- 断电计数器:固定16秒,默认启用,可以禁用,复位后产生IPP_WDOG信号.(任何复位都会重新启用计数器)
- 虽然只是一只看门狗,但是却有外部引脚,让狗咬你MCU时候同时复位外面其他器件.
    - ANY 引脚,任意一只狗咬人都输出.
    - B 引脚,单独某只狗咬人输出.
    - RST_B_DEB 引脚,狗咬内部复位就输出.



复位原因(WRSR),既然看门狗主要作用是复位,但是上电POR也是复位,不同的复位状态可能对应不同的复位逻辑,所以还是区分一下比较好~

- POR - 上电复位,人人都知道了,不解释了.
- TOUT - 狗超时复位,也是人人都知道了.
- SFTW - 软件复位,以前设计软件是否需要重启设备,经常就是开狗不喂狗,需要等待,现在都有一条指令直接复位了.(NVIC复位控制表示不服气)

```
1 switch (WDOG1->WRSR &(kWDOG_PowerOnResetFlag | kWDOG_TimeoutResetFlag | k
WDOG_SoftwareResetFlag))
2 {
3 case kWDOG_PowerOnResetFlag:
4   break;
5 case kWDOG_TimeoutResetFlag:
6   break;
```

```
case kWDOG_SoftwareResetFlag:
  break;
default:
   break;
}
```

示例实现

- WDOG_WICR_WICT(x) => x是0.5秒,多少x就是多少个0.5秒,设定1是0.5秒,设定2是1秒.
- WDOG_WCR_WT(x) => x是0.5秒,多少x就是多少个0.5秒,但是基础有0.5秒,设定1就是1秒,设定2是1.5秒.
- WDOG_WMCR_PDE(x) => 掉电模式使能
- WDOG_WSR => 0x5555,0xAAAA喂狗.
- 如果忘记喂狗,或者在DEBUG模式中FREEZE,就有可能被狗咬了.

```

volatile uint8_t uIntFlag = 0;
volatile uint8_t FeedCnt = 0;

void WDOG1_IRQHandler(void)
{
  WDOG1->WICR |= WDOG_WICR_WTIS(1);
  /* User code. User can do urgent case before timeout reset.
     * IE. user can backup the ram data or ram log to flash.
      * the period is set by config.interruptTimeValue, user need to
      * check the period between interrupt and timeout.
      */
  uIntFlag = 1;
}

static void MainTask(void *pvParameters)
{
  /* Set configuration */
  CLOCK_EnableClock(kCLOCK_Wdog1);

  WDOG1->WICR = WDOG_WICR_WICT(4) | WDOG_WICR_WIE(1);
  WDOG1->WMCR = WDOG_WMCR_PDE(0);
  WDOG1->WCR = WDOG_WCR_WDE(1) | WDOG_WCR_WT(15) | WDOG_WCR_SRS(1) | WDOG_WCR_WDA(1);

```

```
  EnableIRQ(WDOG1_IRQn);
  for (;;)
  {
    vTaskDelay(pdMS_TO_TICKS(10));
    if (uIntFlag == 1)
    {
      WDOG1->WSR = 0x5555;
      WDOG1->WSR = 0xAAAA;
      FeedCnt++;
      uIntFlag = 0;
    }
  }
}
```

主动复位(一执行就被狗咬):

```
WDOG1->WCR &= ~WDOG_WCR_SRS(1);
```

两个看门狗的区别:

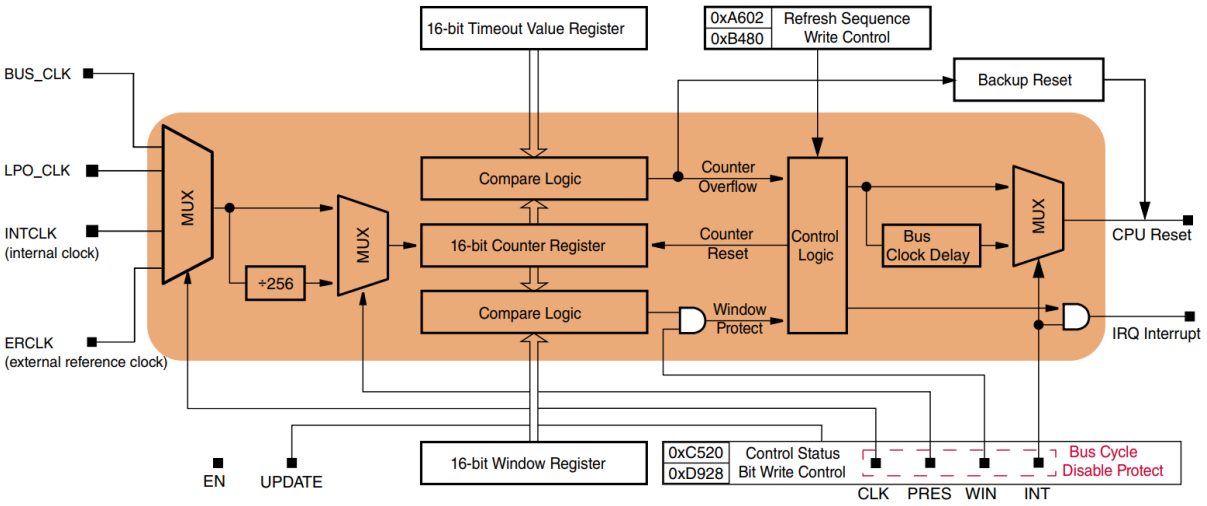
- WDOG1不喂狗发送系统复位.
- WDOG2不喂狗发送SNVS中断.(用于TrustZone,但是M7似乎是没有的,当白送看门狗了)

RTWDOG (WDOG3)

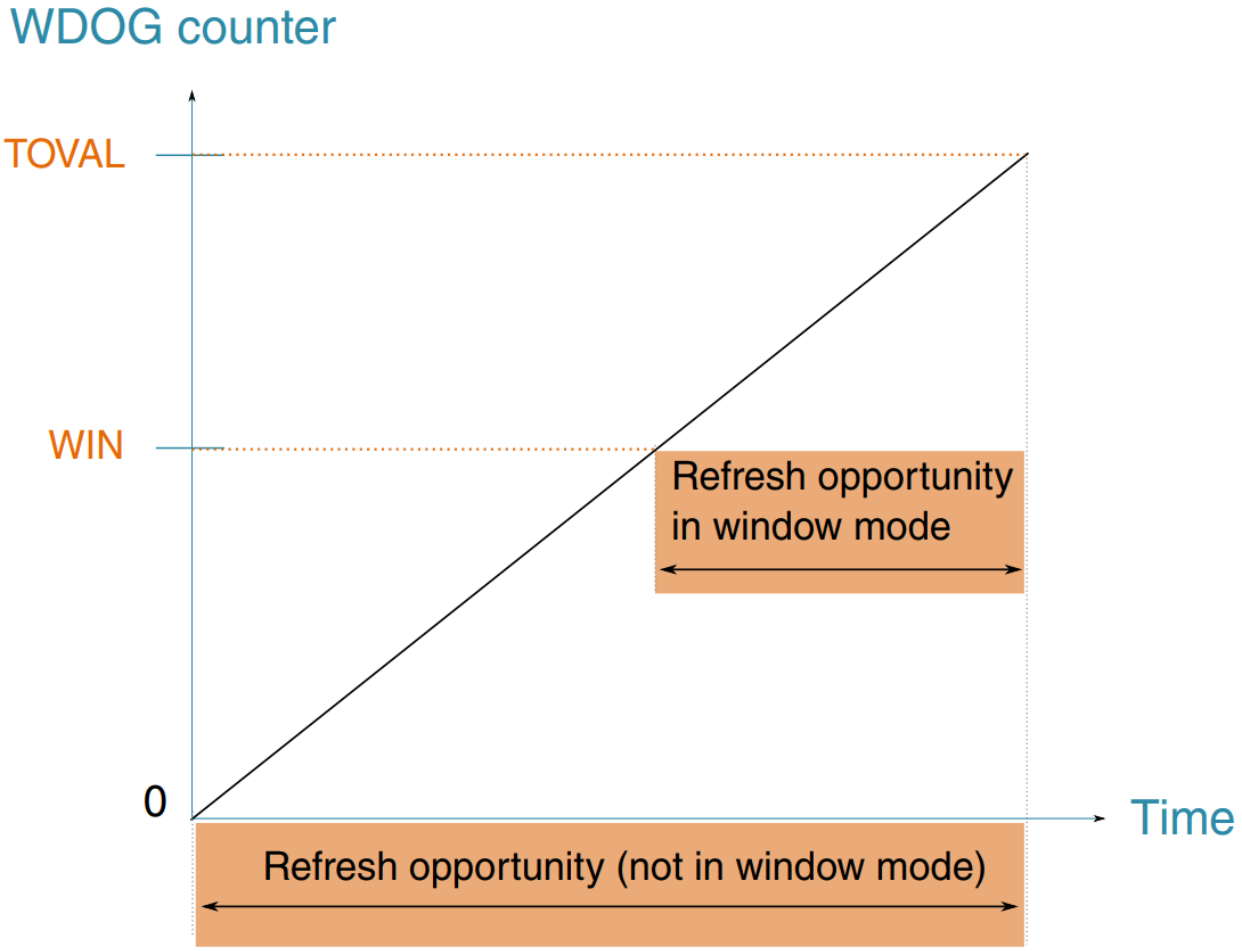
独立看门狗,时钟什么都是独立的,可以说复杂了不少啊,简直是一个全能型的看门狗.

- 可选输入时钟
  - BUS_CLK(低速)
  - LPO_CLK(默认)
  - INTCLK(内部时钟)
  - ERCLK(外部时钟)
- 可编程超时(16位,固定256倍分频,可旁路分频器,即不分频.)
- 喂狗序列:0xA602 -> 0xB480
- 窗口模式:只能在窗口期喂狗,16位.
- 输出ISR中断,在复位前发生中断,中断后的255总线周期会复位CPU,如果CS[INT]没使能中断,则不会延迟255周期.(似乎灵活性差一些啊,而且实在是干不了什么,特别是调试时候,保存都来不及查看就复位了...)
- 配置需要一次性写入,配置寄存器有锁,重配置需要CS[UPDATE]为1,否则只能复位重配置,MCU开机后等待一会再配置,因为WDOG3时钟滞后于系统主时钟.(当然先干别的事情也行)

- 备份重置(安全措施):如果主时钟丢失,无法监视WDOG时钟,并且WDOG发生2次溢出,会生成复位事件,如果这个时候需要发起ISR,那么ISR退出后再溢出会直接复位MCU.
- 测试功能:CS[TST] 区分测试模式和用户功能,但是实际用途当然还是用户模式,因为RTWDOG由16位辣么长,时间比较长才设计这样的东西的.
- 关闭WDOG:关中断,解锁寄存器,关闭,再开中断(按需)



可能有人不知道什么叫窗口模式的狗,这个图解释得很好,有窗口时,太早喂狗是不可以得.



示例实现

- 开启中断实现
- 配置窗口模式
- 喂狗6次,然后放任狗随便干.

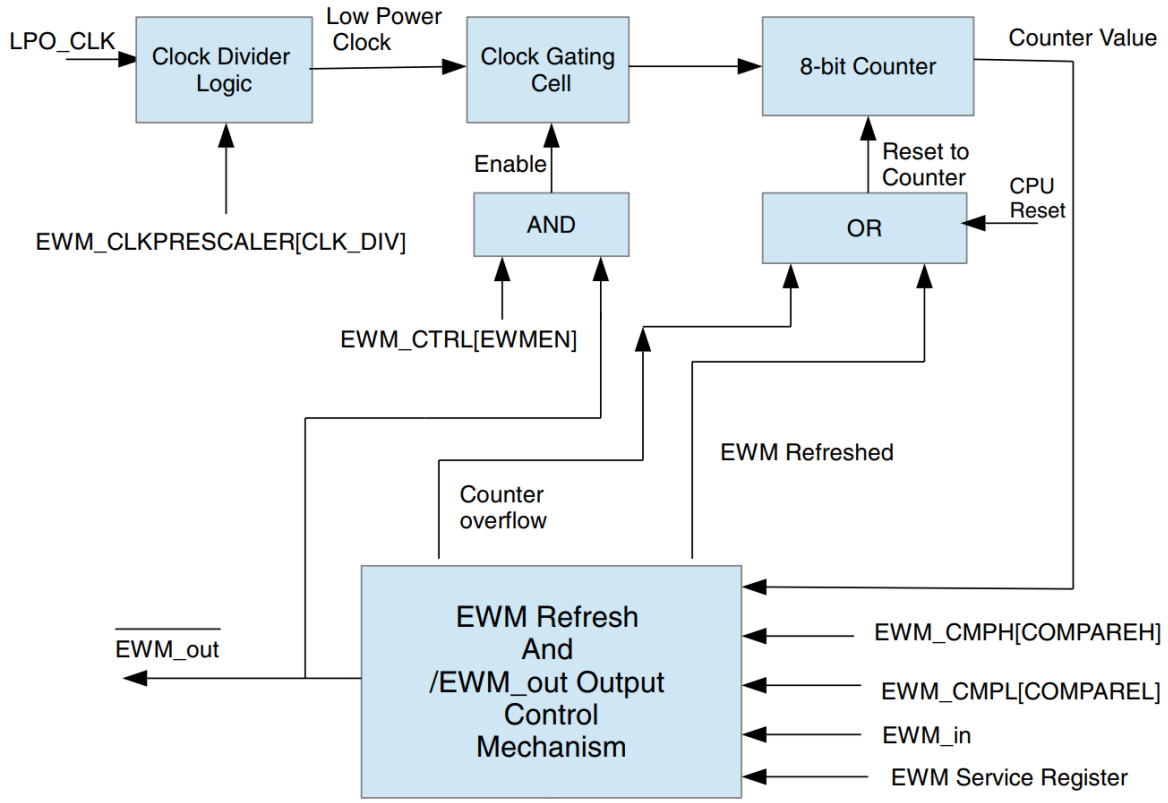
```
volatile uint32_t primaskValue = 0U;

/* 只有在没有喂狗时候,超时那一刻才会发生,调试状态下无法截获. */
void RTWDOG_IRQHandler(void)
{
  RTWDOG->CS |= kRTWDOG_InterruptFlag;
  __DSB();
}

 static void MainTask(void *pvParameters)
 {
   CLOCK_EnableClock(kCLOCK_Wdog3);

   primaskValue = DisableGlobalIRQ();

   RTWDOG->CNT = 0xD928C520U;

   while ((RTWDOG->CS & RTWDOG_CS_ULK_MASK) == 0)
   {
   }

   RTWDOG->WIN = 300;
   RTWDOG->TOVAL = 600;
   RTWDOG->CS = RTWDOG_CS_EN(1) | RTWDOG_CS_CLK(kRTWDOG_ClockSource1) |
                RTWDOG_CS_INT(1) | RTWDOG_CS_WIN(1) | RTWDOG_CS_UPDATE(1)
                RTWDOG_CS_WAIT(1) | RTWDOG_CS_PRES(kRTWDOG_ClockPrescaler
Divide256) |
                RTWDOG_CS_CMD32EN(1) | RTWDOG_CS_TST(kRTWDOG_UserModeEnab
led);

   while ((RTWDOG->CS & RTWDOG_CS_RCS_MASK) == 0)
   {
   }

```

```
  EnableIRQ(RTWDOG_IRQn);
  EnableGlobalIRQ(primaskValue);

  /* 在窗口期(300 - 600)喂狗6次. */
  for (uint8_t i = 0; i < 6; i++)
  {
    if (400 < RTWDOG->CNT)
    {
      primaskValue = DisableGlobalIRQ();
      RTWDOG->CNT = 0xB480A602U;
      EnableGlobalIRQ(primaskValue);
    }
  }

  for (;;)
  {
  }
}
```

External Watchdog Monitor (EWM)

这是一只监视外部电路的看门狗,他主要用来单纯的复位外部的设备,大概由下面的东西构成:

- 独立时钟:LPO_CLK(可以分频)
- 可编程超时.
- 窗口看门狗能力.
- 严格的喂狗机制:必须63个周期内喂完.
- 输出输入端口(IN/OUT)
  - OUT引脚用来复位别人,当你没喂狗时候,OUT就会输出了~ (这个狗输出IO操作,其他狗都是咬死MCU的,真是奇怪.)
  - IN引脚接受外部芯片的FAULT信号,比如电源芯片FAULT了,可以给他输出个信号,接着EWM就发生中断,然后OUT也输出信号复位其他人.
  - 当然都需要使能.
  - 当然IO也需要自己配置.(其实可以生成啦~)
- 可以发起中断(INTEN)
- 等待,停止,掉电状态下,OUT引脚高阻.



98这个看门狗太简单,以至于不知道说什么好,最主要是用于外围电路的,另外这个狗没中断标志,发生中断干什么都不影响狗自身的操作.

```
volatile uint32_t primaskValue = 0U;

void EWM_IRQHandler(void)
{
 __DSB();
}

static void MainTask(void *pvParameters)
{
  /* Set configuration */
  CLOCK_EnableClock(kCLOCK_Ewm0);

  EWM->CLKPRESCALER = 127;
  EWM->CLKCTRL = kEWM_LpoClockSource0;
  EWM->CMPL = 0x55;
  EWM->CMPH = 0xAA;
  EWM->CTRL = EWM_CTRL_EWMEN(1) | EWM_CTRL_INTEN(1);
  NVIC_EnableIRQ(EWM_IRQn);

  for (;;)
  {
```

```
22    vTaskDelay(pdMS_TO_TICKS(10));
23    primaskValue = DisableGlobalIRQ();
24    EWM->SERV = 0xB4;
25    EWM->SERV = 0x2C;
26    EnableGlobalIRQ(primaskValue);
27    }
28  }
```

各位大佬,这一篇废话算是写完了,最好还是结合手册多看多练,也欢迎到我网站(www.taterli.com)转转,暂时就先这样吧.