1. HET 时基概念

1.1 HET 各时钟频率

Table 19-5. Prescale Factor Register Encoding

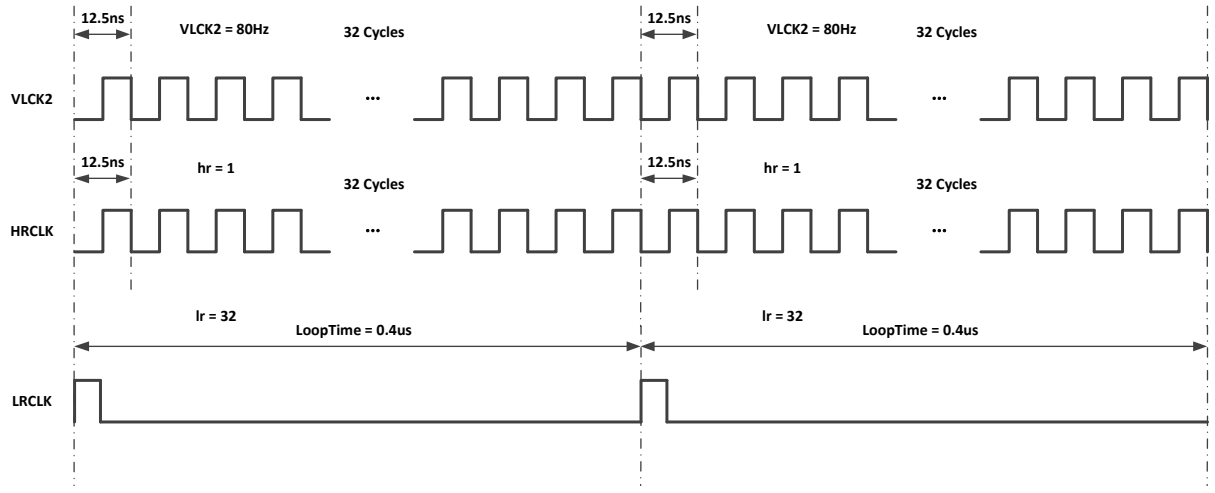
LRPFC - Loop Resolution		HRPFC - High Resolution	
HETPFR[10:8]	Prescale Factor lr	HETPFR[5:0]	Prescale Factor hr
000	/1	000000	/1
001	/2	000001	/2
010	/4	000010	/3
011	/8	000011	/4
100	/16	: : : : :	:
101	/32	111101	/62
110	/64	111110	/63
111	/128	111111	/64

以下具体数据以 VCLK2=80MHz，hr=1，lr=32 为例。

当设置 VCLK2=80MHz，High Resolution Prescale 为 00000(B)，Loop Resolution Prescale 为 101(B)，则

$$\text{LoopTime} = \frac{\text{hr} \times \text{lr}}{\text{VCLK2}} = \frac{1 \times 32}{80\text{M}} = 0.4\ \mu\text{s}$$

在 lr=32 时，指令中的 hr_data 部分（7bit）的最后 2 个 bit 是无效的，即仅有高 5 位数据有效。但 PWM 控制精度仍为 1/HRCLK（此例中为 1/80MHz=12.5ns）。

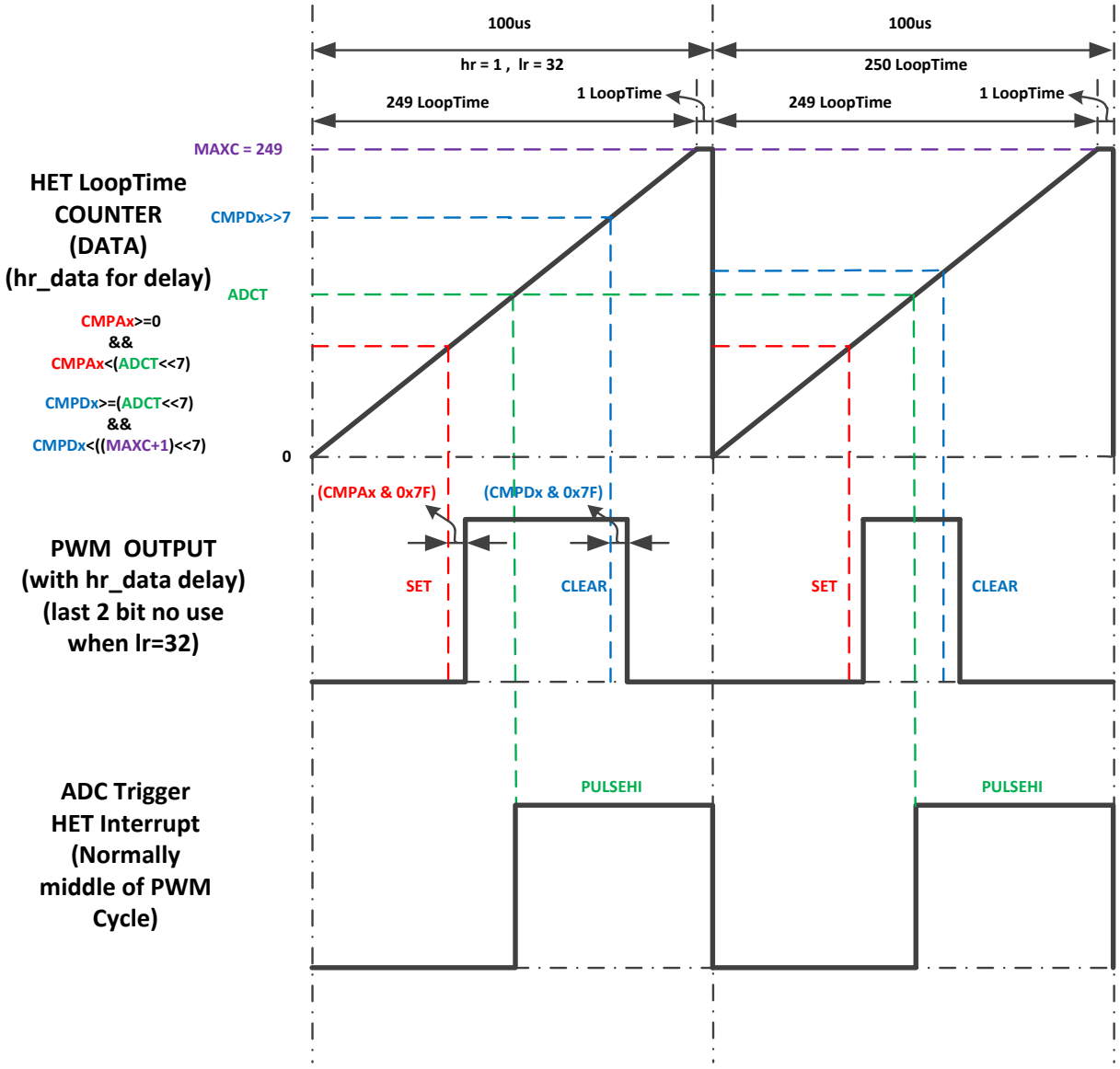


2. PWM 产生原理与实现

2.1 PWM 产生原理

实现 10kHz 的 PWM，其周期为 100μs，需要的 LoopTime 数为

$$N = \frac{100\mu s}{0.4\mu s} = 250$$



HET LoopTime counter 波形如上图所示为锯齿波（实际上是非连续的阶梯，最后一个 LRP 用阶梯表示），其计数从 0~249(MAXC)，每个 counter 持续 1LR 周期。在本例中为 32 个 HR 周期，0.4μs。在 HET 的指令中，LoopTime counter 的数值正好是用于匹配的 DATA 部分（高 25 位）。

- MAXC：PWM 周期控制变量，其值为一个 PWM 所需 LoopTime 周期数减 1。
- ADCT：PWM 半周期控制变量，用于前后两个半个周期匹配指令跳转控制（详见 HET 指令 TRIG1）。同时，作为 ADC 触发源（TRIG2）和 HET 中断源（INT0）。其值为 PWM 周期所需 LoopTime 周期数的一半。在本例中为 125。由于匹配指令仅匹配高 25 位，所以当 LoopTime counter 值达到 125 且 hr_data=0 时即开始发生匹配，正好半周期。
- CMPAx：PWM 前半周期匹配控制变量，用于 PWM 前半周期电平控制。当 LoopTime counter 值与 CMPAx 高 25 位发生匹配时，延时 CMPAx 低 7 位有效位数值（本例中为 5），即 hr_data 有效位数值个 HRT 后，引脚电平动作（SET 或 CLEAR）。
- CMPDx：PWM 后半周期匹配控制变量，用于 PWM 后半周期电平控制。实现与 CMPAx 相同。

2.2 HET 指令实现

1. 变量定义

```
MAXC        .equ  249           ; Default PWM base period
ADCT        .equ  125           ; Default compare value for ADC trigger pulse generation

Am1A        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm1A        .equ  125           ; Default compare value (will be overwritten by CPU)

Am2A        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm2A        .equ  125           ; Default compare value (will be overwritten by CPU)

Am3A        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm3A        .equ  125           ; Default compare value (will be overwritten by CPU)

Am1B        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm1B        .equ  125           ; Default compare value (will be overwritten by CPU)

Am2B        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm2B        .equ  125           ; Default compare value (will be overwritten by CPU)

Am3B        .equ  125           ; Default compare value (will be overwritten by CPU)
Cm3B        .equ  125           ; Default compare value (will be overwritten by CPU)
```

其中，AmxA 与 CMPAx 对应，CmxA 与 CMPDx。

2. PWM 引脚定义

```
PWM_1A    .equ  0               ; PWM signal 1A, PWM signal for high-side FET of phase U
PWM_2A    .equ  2               ; PWM signal 2A, PWM signal for high-side FET of phase V
PWM_3A    .equ  4               ; PWM signal 3A, PWM signal for high-side FET of phase W

PWM_1B    .equ  1               ; PWM signal 1B, PWM signal for low-side FET of phase U
PWM_2B    .equ  3               ; PWM signal 2B, PWM signal for low-side FET of phase V
PWM_3B    .equ  5               ; PWM signal 3B, PWM signal for low-side FET of phase W
```

```
INT_TRIG2 .equ  1               ; HET pin for ADC trigger
```

引脚定义可以根据实际情况进行更改。

需要注意的是，ADC 触发源引脚必须定义为能够触发 ADC 的引脚。具体引脚号参见 Datasheet。

3. 实现指令

```
Lm00     CNT     {next= Lm11, reg = T, max= MAXC, irq = OFF,data=0}  ; Counter -> Periode / Value on REG T
```

更新 LoopTime Counter 到 T 寄存器，每个 LoopTime 执行一次。

```
Lm21     BR      { next= TRIG0, cond_addr= Lm12, event = ZERO}         ; Branch for Updating Signal
Lm22     DJZ     { next=UPD_Am1A,cond_addr=TRIG0,reg=A,irq=OFF,data=0} ;To Guarantee Integrity of updating
data.
```

控制更新 PWM 匹配值更新指令跳转。

```
UPD_Am1A   MOV32 { next=UPD_Cm1A,remote=Lm15,type=IMTOREG&REM,reg=A,data=Am1A,hr_data=0};
UPD_Cm1A   MOV32 { next=UPD_Am1B,remote=Lm16,type=IMTOREG&REM,reg=A,data=Cm1A,hr_data=0};

UPD_Am1B   MOV32 { next=UPD_Cm1B,remote=Lm45,type=IMTOREG&REM,reg=A,data=Am1B,hr_data=0};
UPD_Cm1B   MOV32 { next=UPD_Am2A,remote=Lm46,type=IMTOREG&REM,reg=A,data=Cm1B,hr_data=0};

UPD_Am2A   MOV32 { next=UPD_Cm2A,remote=Lm25,type=IMTOREG&REM,reg=A,data=Am2A,hr_data=0};
UPD_Cm2A   MOV32 { next=UPD_Am2B,remote=Lm26,type=IMTOREG&REM,reg=A,data=Cm2A,hr_data=0};

UPD_Am2B   MOV32 { next=UPD_Cm2B,remote=Lm55,type=IMTOREG&REM,reg=A,data=Am2B,hr_data=0};
UPD_Cm2B   MOV32 { next=UPD_Am3A,remote=Lm56,type=IMTOREG&REM,reg=A,data=Cm2B,hr_data=0};

UPD_Am3A   MOV32 { next=UPD_Cm3A,remote=Lm35,type=IMTOREG&REM,reg=A,data=Am3A,hr_data=0};
UPD_Cm3A   MOV32 { next=UPD_Am3B,remote=Lm36,type=IMTOREG&REM,reg=A,data=Cm3A,hr_data=0};

UPD_Am3B   MOV32 { next=UPD_Cm3B,remote=Lm65,type=IMTOREG&REM,reg=A,data=Am3B,hr_data=0};
UPD_Cm3B   MOV32 { next=INT0,remote=Lm66,type=IMTOREG&REM,reg=A,data=Cm3B,hr_data=0};
```

更新 PWM 匹配值。

```
INT0     ECMP    { next= IND2, cond_addr= IND2, hr_lr=LOW, en_pin_action=OFF, pin=INT_TRIG2, reg= T,
data=ADCT, hr_data=0};
```

匹配触发 HET 中断。

```
TRIG1    MCMP    { next= Lm15, cond_addr= Lm16, hr_lr=HIGH, en_pin_action=ON, pin=INT_TRIG0,
order=REG_GE_DATA, action=PULSEHI, reg= T, data=ADCT, hr_data=0}
```

匹配控制 PWM 匹配指令跳转。

当 T<ADCT，跳转到 lm15；

当 T>=ADCT，跳转到 lm16。

```
;---------------------------------------- Motor PWM Pair 1A----------------------------
Lm15     ECMP    { next= Lm25, cond_addr= Lm25, hr_lr=HIGH, en_pin_action=ON, pin=PWM_1A, action=SET, reg=
T, data=Am1A, hr_data=0};
;---------------------------------------- Motor PWM Pair 2A----------------------------
Lm25     ECMP    { next= Lm35, cond_addr= Lm35, hr_lr=HIGH, en_pin_action=ON, pin=PWM_2A, action=SET, reg=
T, data=Am2A, hr_data=0};
;---------------------------------------- Motor PWM Pair 3A----------------------------
Lm35     ECMP    { next= Lm45, cond_addr= Lm45, hr_lr=HIGH, en_pin_action=ON, pin=PWM_3A, action=SET, reg=
T, data=Am3A, hr_data=0};
```

```
;----------------------------------------- Motor PWM Pair 1B----------------------------
Lm45     ECMP    { next= Lm55, cond_addr= Lm55, hr_lr=HIGH, en_pin_action=ON, pin=PWM_1B, action=CLEAR,
reg= T, data=Am1B, hr_data=0};
;----------------------------------------- Motor PWM Pair 2B----------------------------
Lm55     ECMP    { next= Lm65, cond_addr= Lm65, hr_lr=HIGH, en_pin_action=ON, pin=PWM_2B, action=CLEAR,
reg= T, data=Am2B, hr_data=0};
;----------------------------------------- Motor PWM Pair 3B----------------------------
Lm65     ECMP    { next= Lm00, cond_addr= Lm00, hr_lr=HIGH, en_pin_action=ON, pin=PWM_3B, action=CLEAR,
reg= T, data=Am3B, hr_data=0};
```

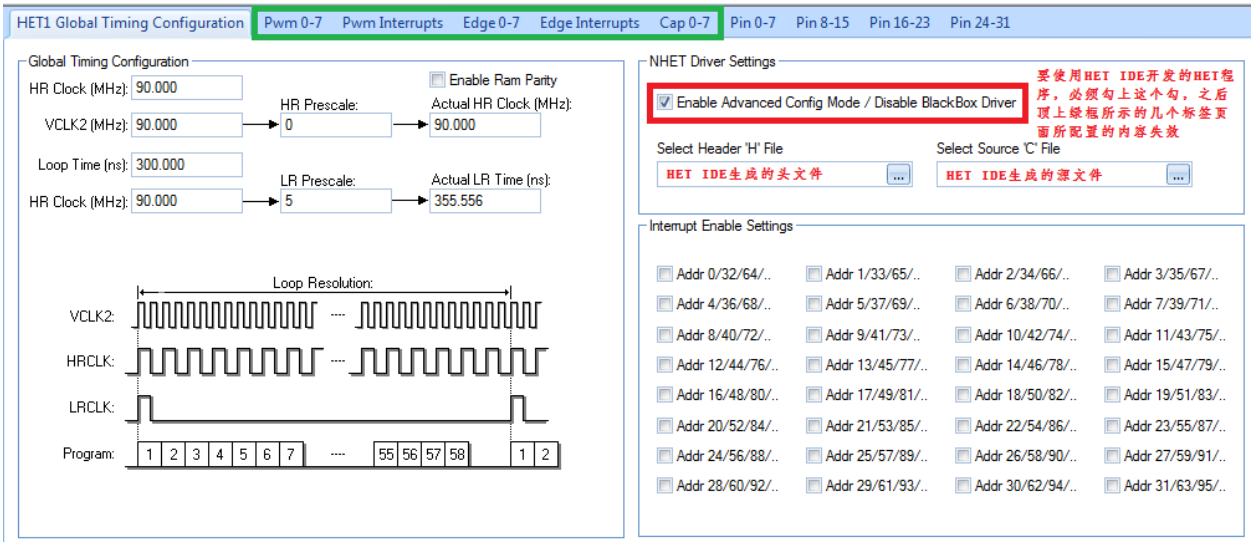
前半周期 PWM 匹配。

```
;----------------------------------------- Motor PWM Pair 1A----------------------------
Lm16     ECMP    { next= Lm26, cond_addr= Lm26, hr_lr=HIGH, en_pin_action=ON, pin=PWM_1A, action=CLEAR,
reg= T, data=Cm1A, hr_data=0}
;----------------------------------------- Motor PWM Pair 2A----------------------------
Lm26     ECMP    { next= Lm36, cond_addr= Lm36, hr_lr=HIGH, en_pin_action=ON, pin=PWM_2A, action=CLEAR,
reg= T, data=Cm2A, hr_data=0}
;----------------------------------------- Motor PWM Pair 3A----------------------------
Lm36     ECMP    { next= Lm46, cond_addr= Lm46, hr_lr=HIGH, en_pin_action=ON, pin=PWM_3A, action=CLEAR,
reg= T, data=Cm3A, hr_data=0}
;----------------------------------------- Motor PWM Pair 1B----------------------------
Lm46     ECMP    { next= Lm56, cond_addr= Lm56, hr_lr=HIGH, en_pin_action=ON, pin=PWM_1B, action=SET, reg=
T, data=Cm1B, hr_data=0}
;----------------------------------------- Motor PWM Pair 2B----------------------------
Lm56     ECMP    { next= Lm66, cond_addr= Lm66, hr_lr=HIGH, en_pin_action=ON, pin=PWM_2B, action=SET, reg=
T, data=Cm2B, hr_data=0}
;----------------------------------------- Motor PWM Pair 3B----------------------------
Lm66     ECMP    { next= Lm00, cond_addr= Lm00, hr_lr=HIGH, en_pin_action=ON, pin=PWM_3B, action=SET, reg=
T, data=Cm3B, hr_data=0}
```

后半周期 PWM 匹配。

2.3 C 语言接口

2.3.1 HET 程序初始化



HET 程序在上电后需要 Copy 到 HET RAM 中。

现在 Halcogen 已经提供了 HET IDE 与 Halcogen 的接口，如上图。
使用 Halcogen 提供的这个接口，在调用 hetInit()函数后，HET 程序就自动 Copy 到了 HET RAM 中。
HET 程序一旦 Copy 到 HET RAM 中就开始运行。

2.3.2 pwm.h

1. PWM 相关控制变量定义

```
#define vCNT_MAX                    249
#define HALF_PERIOD                 (125<<7)
#define PWM_BASE_ADDR               3                   // PWM control base index
#define HET_INT_ADDR                15                  // HET interrupt
instruction index

#define HET_FREQUENCY               90                  // HET module clock
#define HET_HR                      1                   // HR resolution
#define HET_LR                      32                  // LR resolution
#define HET_LR_REG                  5                   // LR resolution register
setting
#define ISR_FREQUENCY     (1000*HET_FREQUENCY/(HET_HR*HET_LR*(vCNT_MAX+1)))    // HET
interrupt frequency
#define BASE_FREQ                   120                 // Sine wave base
frequency
#define T                       (0.001/ISR_FREQUENCY)       // parameter
```

2.3.3 pwm_drv.h

1. 最大 HR 周期计数值定义

```
(128*(vCNT_MAX + 1))-1              /*PeriodMax*/
```

由于 ECMP 和 MCMP 指令仅匹配前 25 位，即 DATA 部分。以 hr=1,lr=32 为例，CMPAx 和 CMPDx 的取值范围为

$$0 \le CMPAx \le 7999$$

$$8000 \le CMPDx \le 15999$$

才能使

$$0 \le (CMPAx>>7) \le 124$$

$$125 \le (CMPDx>>7) \le 249$$

2.4 pwm.c

1. void het_int_enable(void)

- 选择 het 中断源指令

```
hetRAM1->Instruction[HET_INT_ADDR%32].Control |= 0x01;
```

第 HET_INT_ADDR 条指令即（INT0），该指令 Control 部分的最后一位为中断使能位。若该位为“1”，表示该指令条件匹配时产生 het 中断。

HET 指令的中断一共只有 32 个，第 x 条指令与 x+32*n 条指令共享一个中断源。因此，HET_INT_ADDR 需要以 32 取模，**在 HET 程序设计中，应尽量避免两条指令产生同一个中断源。**

➢ 使能 HET 中断

```
hetREG1->INTENAS = (1<<(HET_INT_ADDR%32));
```
使能 HET 中断寄存器 INTENAS 中的 HET_INT_ADDR 位。

2. `void het_pwm_update(float Ta, float Tb, float Tc, int deadzone)`

➢ 死区时间计算

输入的 deadzone 的单位为 **ns**。

需要计算最终加入到匹配值（CMPDx）中的 pwm_drv1.deadzone 值

$$\text{pwm\_drv1.deadzone} = \frac{deadzone \times HET\_FREQUENCY}{1000 \times HET\_HR} \ll HET\_LR$$

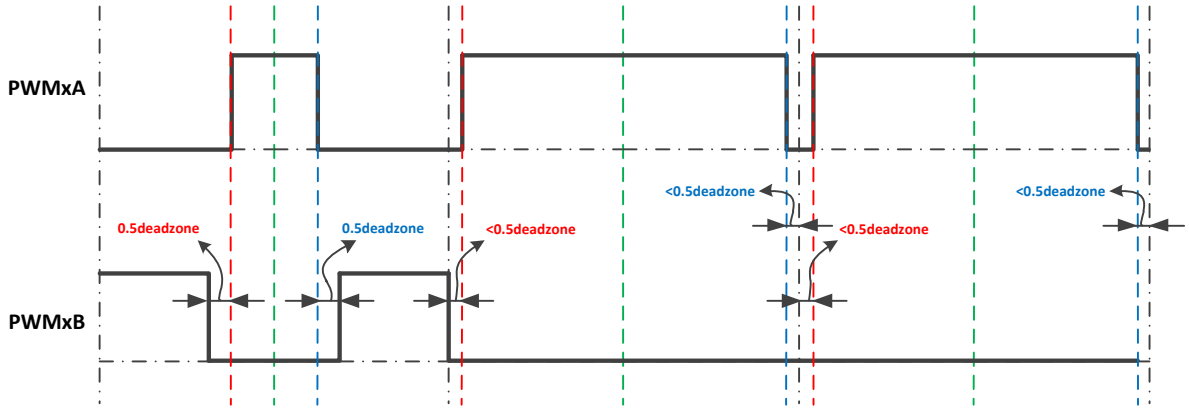
➢ 上桥 CMPAx /CMPDx 更新

```
hetRAM1->Instruction[PWM_BASE_ADDR].Data      = pwm_drv1.CMPA1;   // PWM-1A
hetRAM1->Instruction[PWM_BASE_ADDR+1].Data    = pwm_drv1.CMPD1;
hetRAM1->Instruction[PWM_BASE_ADDR+4].Data    = pwm_drv1.CMPA3;   // PWM-2A
hetRAM1->Instruction[PWM_BASE_ADDR+5].Data    = pwm_drv1.CMPD3;
hetRAM1->Instruction[PWM_BASE_ADDR+8].Data    = pwm_drv1.CMPA5;   // PWM-3A
hetRAM1->Instruction[PWM_BASE_ADDR+9].Data    = pwm_drv1.CMPD5;
```

➢ 下桥 CMPDAx/CMPDx 更新

下桥 CMPDAx/CMPDx 更新需计算死区时间。

由于 PWM 按周期中间对称设计，所以死区时间也以周期中点左右对称加入。如下图所示。



$$\begin{cases} CMPA_{x+1} = 0, & CMPA_x < pwm\_drv1.deazone/2 \\ CMPA_{x+1} = CMPA_x - pwm\_drv1.deazone/2, & CMPA_x > pwm\_drv1.deazone/2 \end{cases}$$

$$\begin{cases} CMPD_{x+1} = CMPD_x + pwm\_drv1.deazone/2, & CMPD_x + pwm\_drv1.deazone/2 < pwm\_drv1.PeriodMax \\ CMPD_{x+1} = pwm\_drv1.PeriodMax + 1, & CMPD_x + pwm\_drv1.deazone/2 > pwm\_drv1.PeriodMax \end{cases}$$

注：$CMPD_{x+1}$=pwmdrv1.PeriodMax+1，只有这样才能使 $CMPD_{x+1}$=16000（按本例设置），即（$CMPD_{x+1}$=250<<7），在 PWM 周期中不再会发生匹配，保持当前引脚电平状态。

```
if(pwm_drv1.CMPA1 < pwm_drv1.deadzone/2)                    // PWM-1B
      hetRAM1->Instruction[PWM_BASE_ADDR+2].Data = 0;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+2].Data = pwm_drv1.CMPA1 -
pwm_drv1.deadzone/2;
if(pwm_drv1.CMPD1+pwm_drv1.deadzone/2 > pwm_drv1.PeriodMax)
      hetRAM1->Instruction[PWM_BASE_ADDR+3].Data = pwm_drv1.PeriodMax + 1;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+3].Data = pwm_drv1.CMPD1 +
pwm_drv1.deadzone/2;

if(pwm_drv1.CMPA3 < pwm_drv1.deadzone/2)                    // PWM-2B
      hetRAM1->Instruction[PWM_BASE_ADDR+6].Data = 0;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+6].Data = pwm_drv1.CMPA3 -
pwm_drv1.deadzone/2;
if(pwm_drv1.CMPD3+pwm_drv1.deadzone/2 > pwm_drv1.PeriodMax)
      hetRAM1->Instruction[PWM_BASE_ADDR+7].Data = pwm_drv1.PeriodMax + 1;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+7].Data = pwm_drv1.CMPD3 +
pwm_drv1.deadzone/2;

if(pwm_drv1.CMPA5 < pwm_drv1.deadzone/2)                    // PWM-3B
      hetRAM1->Instruction[PWM_BASE_ADDR+10].Data = 0;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+10].Data = pwm_drv1.CMPA5 -
pwm_drv1.deadzone/2;
if(pwm_drv1.CMPD5+pwm_drv1.deadzone/2 > pwm_drv1.PeriodMax)
      hetRAM1->Instruction[PWM_BASE_ADDR+11].Data = pwm_drv1.PeriodMax + 1;
else
      hetRAM1->Instruction[PWM_BASE_ADDR+11].Data = pwm_drv1.CMPD5 +
```

```
pwm_drv1.deadzone/2;
```

- 匹配值更新

```
hetRAM1->Instruction[PWM_BASE_ADDR-1].Data = 1<<7;
```

只有当 HET 的第 PWM_BASE_ADDR-1 条指令的 DATA 部分写入 1 之后，再下一个 LoopTime 周期中，PWM 的匹配值才能得到更新。

3. QEP 计数原理与实现

3.1 QEP 计数原理

1）正向计数——加

QEPA	QEPB	QEPI
Fall	Low	No edge
Rise	High	No edge
High	Fall	No edge
Low	Rise	No edge

2）反向计数——减

QEPA	QEPB	QEPI
Fall	High	No edge
Rise	Low	No edge
Low	Fall	No edge
High	Rise	No edge

3）Index 清零

QEPA	QEPB	QEPI
Don’t care	Don’t care	Rise

- 正向计数，当计数值到最大值时，清零。
- 反向计数，当计数值到零值时，装载最大值。

3.2 HET 指令实现

1. 变量定义

```
QEP_MAX   .equ  7999            ; Default QEP encoder max count
```

2. 引脚定义

```
Pin_A2     .equ  24             ; Signal A from QEP encoder
Pin_B2     .equ  13             ; Signal B from QEP encoder
INDEX2     .equ  15             ; Index from QEP encoder
```

3. 指令实现

1）正/反向判断

```
;---------------------------check: aFbL  /   aFbH ------------------------------
AFE2     BR      { next = ARE2, cond_addr = AFL2, event = Fall, pin = Pin_A2}
AFL2     BR      { next = BAC2, cond_addr = FOR2, event = Low, pin = Pin_B2}
;---------------------------check: aRbL  /   aRbH ------------------------------
ARE2     BR      { next = BFE2, cond_addr = ARL2, event = rise, pin = Pin_A2}
ARL2     BR      { next = BAC2, cond_addr = FOR2, event = high, pin = Pin_B2}
;---------------------------check: bFaL  /   bFaH ------------------------------
BFE2     BR      { next = BRE2, cond_addr = BFL2, event = Fall, pin = Pin_B2}
BFL2     BR      { next = BAC2, cond_addr = FOR2, event = high, pin = Pin_A2}
;---------------------------check: bRaL  /   bRaH ------------------------------
BRE2     BR      { next = TRIG2, cond_addr = BRL2, event = rise, pin = Pin_B2}
BRL2     BR      { next = BAC2, cond_addr = FOR2, event = low, pin = Pin_A2}
```

- BAC2：反向计数跳转地址
- FOR2：正向计数跳转地址

2）正向计数

```
FOR2     CNT     { next = TRIG2, reg = NONE, max = QEP_MAX};; count up (+1) ; read result here    //26
```

- CNT 指令用于正向计数，计数值到最大值时，自动回零

3）反向计数

```
BAC2     MOV32   { next = LIM2, remote = FOR2, type = REMTOREG, reg = A}
LIM2     ECMP    { next= SSUB2, cond_addr= HIL2, hr_lr=LOW, en_pin_action=off, pin=CC27, reg= A, data=0} ;
check if previous value is 0 or not
HIL2     ADM32   { next = TRIG2, remote = FOR2, type = IM&REGTOREM, reg = A, data = QEP_MAX}; 0 -> count
down -> max(QEP_MAX)
SSUB2     ADM32  { next = TRIG2 , remote = FOR2, type = IM&REGTOREM, reg = A, data = 0x1FFFFFF}; count
down(-1)//30
```

- 当前计数值为零值，重新加载最大值（HIL2）
- 当前计数值不为零值，直接减一。用 HET 的加法指令，就是加 0x1FFFFFF（Data 部分为 25 位）

4）Index 清零

```
;--------------------------INDEX-CLR Counter ---------------------------------
IND2     ECNT    { next = AFE2, cond_addr = CLRCNT2, event = Rise, pin = INDEX2, reg = NONE, data =
0};//16
```

➢ 检测 QEPI 引脚上升沿

```
CLRCNT2  MOV32   { next = AFE2, remote = FOR2, type = IMTOREG&REM, reg = A, Data = 0}
```

➢ 把计数值清零

根据前面的描述，QEP 的计数值保存在 FOR2 指令的 Data 部分。

3.3 C 语言接口

1. qep.h

相关变量定义

```
#define      QEP_MAX             7999          // Max count of QEP encoder
#define      Data_Instraction    26            // Index of reading QEP value
```

2. qep.c

QEP 计数值读取函数接口。

```
int ReadEncoder(void)
{
      return (hetRAM1->Instruction[Data_Instraction].Data >> 7);
}
```

读取 FOR2 指令的数据。需要注意的是，只有高 25 位是有效数据，所以需要把得到的 32 位数据右移 7 位。