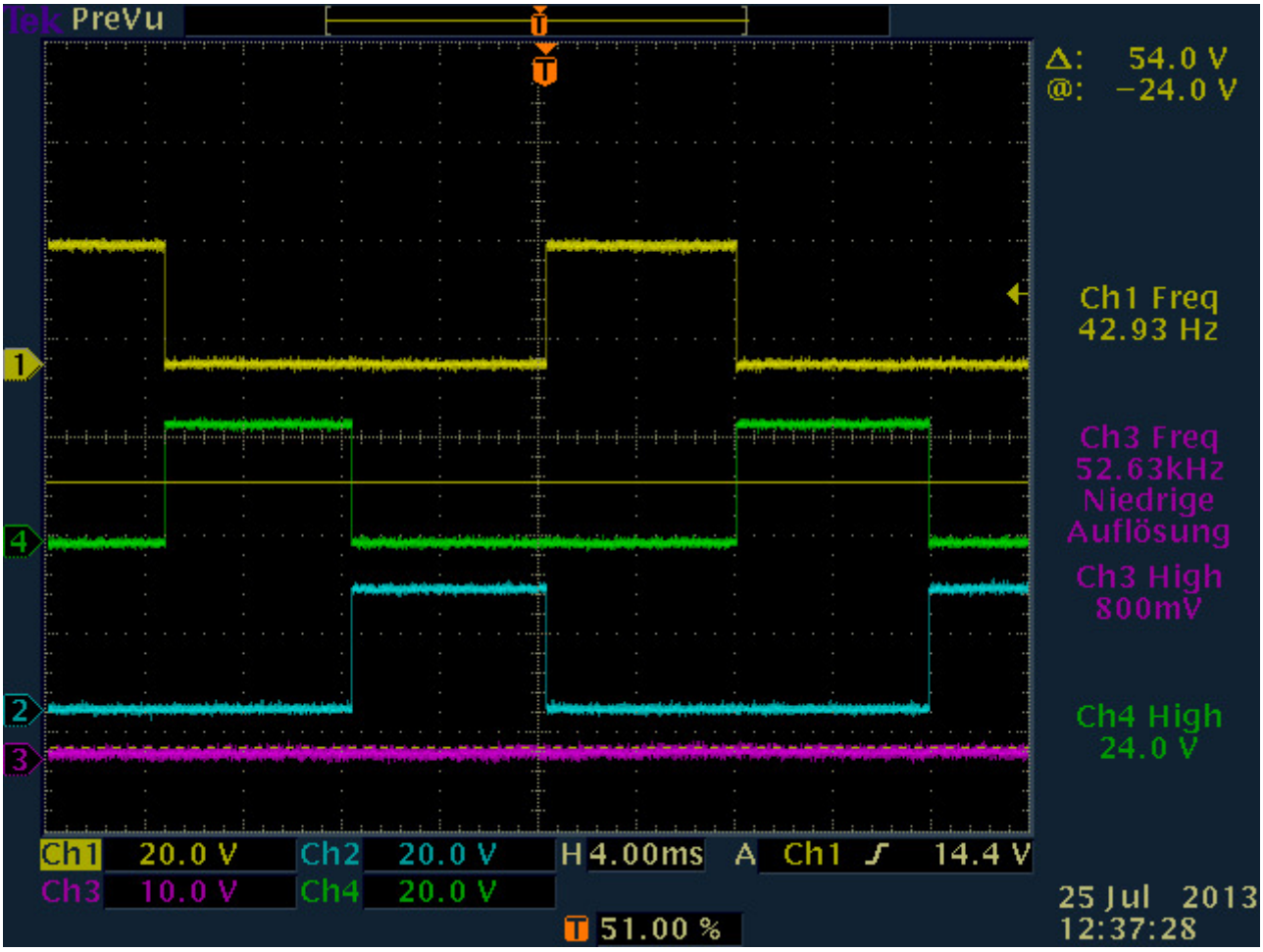
Hello,

i got a Problem with my Motor Control PWM Complementary Outputs. I use for the Motor Control a 6 step PWM Signal interfacing with 3x 120° Hall Sensors. The Signal are 3 Highs and 3 Low PWM's. I Generate the PWMs with a SMT32 MCU "stm32f103CB" and increase the Signals with and 3 Phase Gate Driver from Infineon "6ED003L06-F". My Problem is, i can't Complementary my Low Signals to commutate the motor right. The High Signals are the following pic and the Low Signals are only the Inverted of the High Signal.



The Pins i use are:

/*3Hall signals*/
PB6 -> Hall (1)U
PB7 -> Hall (2)V
PB8 -> Hall (3)W

/*PWM 1-6*/
PB13 -> HS_U
PB14 -> HS_V
PB15 -> HS_W
PA8 -> LS_U
PA9 -> LS_V
PA10 -> LS_W

Now i will to get the right Low Singals the TIM1_CHx (1,2,3) set a Dead Time, but in my code is something wrong, because they does not what i need.

The Code:

```
int main(void)
{
    uint16_t Daumen_value = 0;
```

```
RCC_Configuration1();
/* System Clocks Configuration */
RCC_Configuration();

  GPIO_Configuration();

  Setup_Hall_Sensor();
  //SysTick_Configuration();
  //Setup_ADC1();
/* NVIC Configuration */

  //NVIC_Configuration();
/* GPIO Configuration */

/* SysTick Configuration */
  //SysTick_Configuration();

  TimerPeriod = ((SystemCoreClock/20000));

/* Time Base configuration */
TIM_TimeBaseStructure.TIM_Prescaler = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseStructure.TIM_Period = TimerPeriod;
TIM_TimeBaseStructure.TIM_ClockDivision = 3;
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;

TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);


/* Channel 1, 2,3 and 4 Configuration in PWM mode */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_OutputNState = TIM_OutputNState_Enable;
TIM_OCInitStructure.TIM_Pulse = Channel1Pulse;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_High;
TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Set;
TIM_OCInitStructure.TIM_OCNIdleState = TIM_OCIdleState_Set;

TIM_OC1Init(TIM1, &TIM_OCInitStructure);
  TIM_OC1PreloadConfig(TIM1,ENABLE);

TIM_OC2Init(TIM1, &TIM_OCInitStructure);
  TIM_OC2PreloadConfig(TIM1,ENABLE);

TIM_OC3Init(TIM1, &TIM_OCInitStructure);
  TIM_OC3PreloadConfig(TIM1,ENABLE);


/* Automatic Output enable, Break, dead time and lock configuration*/
TIM_BDTRInitStructure.TIM_OSSRState = TIM_OSSRState_Enable;
TIM_BDTRInitStructure.TIM_OSSIState = TIM_OSSIState_Enable;
TIM_BDTRInitStructure.TIM_LOCKLevel = TIM_LOCKLevel_OFF;
TIM_BDTRInitStructure.TIM_DeadTime = 15;
TIM_BDTRInitStructure.TIM_Break = TIM_Break_Disable;
TIM_BDTRInitStructure.TIM_BreakPolarity = TIM_BreakPolarity_High;
TIM_BDTRInitStructure.TIM_AutomaticOutput = TIM_AutomaticOutput_Enable;
```

```
  TIM_BDTRConfig(TIM1, &TIM_BDTRInitStructure);

  TIM_CCPreloadControl(TIM1, ENABLE);

    TIM_SelectCOM(TIM1, ENABLE);

    TIM_SelectInputTrigger(TIM1, TIM_TS_ITR3);

  TIM_ITConfig(TIM1, TIM_IT_COM, ENABLE);

  /* TIM1 counter enable */
  TIM_Cmd(TIM1, ENABLE);

  /* Main Output Enable */
  TIM_CtrlPWMOutputs(TIM1, ENABLE);

    /* Activate 3Phase Gate Treiber */
    GPIO_WriteBit(GPIOC, GPIO_Pin_13, Bit_SET);


  while (1)
  {

        //if (ADC1ConvertedValue)
        //{
            //TIM1->CCR1 = (int)(ADC1ConvertedValue);
            //TIM1->CCR2 = (int)(ADC1ConvertedValue);
            //TIM1->CCR3 = (int)(ADC1ConvertedValue);
        //}
    }
}

/**
  * @brief  Configures the different system clocks.
  * @param  None
  * @retval None
  */
void RCC_Configuration(void)
{
  /* TIM1, GPIOA, GPIOB, GPIOE and AFIO clocks enable */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_TIM1 | RCC_APB2Periph_GPIOA |
                         RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC |RCC_APB2Periph_AFIO,
ENABLE);

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);

        /* ADCCLK = PCLK2/8 */
  RCC_ADCCLKConfig(RCC_PCLK2_Div8);

  /* Enable DMA1 clock */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);
}

/**
  * @brief  Configure the TIM1 Pins.
  * @param  None
  * @retval None
  */
```

```
void GPIO_Configuration(void)
{
  GPIO_InitTypeDef GPIO_InitStructure;

  /* GPIOA Configuration: Channel 1N, 2N and 3N as alternate function push-pull */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 | GPIO_Pin_10;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  /* GPIOB Configuration: Channel 1, 2 and 3 as alternate function push-pull */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* GPIOC Configuration: Pin 2 as analog input for Daumengas-> ADC */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* 3Phase Gate Treiber */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* Hall 1,2,3 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);

}

/**
  * @brief  Configures the SysTick.
  * @param  None
  * @retval None
  */
void SysTick_Configuration(void)
{
    /* Configure SysTick to generate an interrupt each 100ms (with HCLK = 72MHz) -*/
  /* Select AHB clock(HCLK) as SysTick clock source */
  SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
  /* Setup SysTick Timer for 100 msec interrupts  */
  if (SysTick_Config((SystemCoreClock) / 100))
  {
    /* Capture error */
    while (1);
  }

   NVIC_SetPriority(SysTick_IRQn, 0x1);
}

/**
  * @brief  Configures the nested vectored interrupt controller.
  * @param  None
  * @retval None
  */
```

```
void NVIC_Configuration(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;

#ifdef  VECT_TAB_RAM
  /* Set the Vector Table base location at 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  /* Set the Vector Table base location at 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif

  /* Enable the TIM1 Interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM1_TRG_COM_IRQn;

  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
}

void Setup_ADC1(void)
{

    /* DMA1 Channel1 configuration ----------------------------------------------*/
    ADC_InitTypeDef ADC_InitStructure;
    DMA_InitTypeDef DMA_InitStructure;
    DMA_DeInit(DMA1_Channel1);
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&ADC1->DR;
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)&ADC1ConvertedValue;
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
    DMA_InitStructure.DMA_BufferSize = 1;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
    DMA_InitStructure.DMA_MemoryDataSize = DMA_PeripheralDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
    DMA_Init(DMA1_Channel1, &DMA_InitStructure);

    /* Enable DMA1 channel1 */
    DMA_Cmd(DMA1_Channel1, ENABLE);
    /* ADC1 configuration ------------------------------------------------------*/
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_ScanConvMode = ENABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init(ADC1, &ADC_InitStructure);
    /* ADC1 regular channels configuration */
    ADC_RegularChannelConfig(ADC1, ADC_Channel_2, 1, ADC_SampleTime_13Cycles5 );


    /* Enable ADC1 DMA */
    ADC_DMACmd(ADC1, ENABLE);
    /* Enable ADC1 */
    ADC_Cmd(ADC1, ENABLE);
```

```
    /* Enable ADC1 reset calibration register */
    ADC_ResetCalibration(ADC1);
    /* Check the end of ADC1 reset calibration register */
    while(ADC_GetResetCalibrationStatus(ADC1));
    /* Start ADC1 calibration */
    ADC_StartCalibration(ADC1);
    /* Check the end of ADC1 calibration */
    while(ADC_GetCalibrationStatus(ADC1));
    /* Start ADC1 Software Conversion */
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);
}


void Setup_Hall_Sensor(void)
{

   TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;

  TIM_TimeBaseStructure.TIM_Prescaler = 240;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseStructure.TIM_Period = 65535;
  TIM_TimeBaseStructure.TIM_ClockDivision = 0;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);

    TIM4->CCR2 = 10; // Commutation Delay = 1ms

  // enable hall sensor
  // T1F_ED will be connected to  HallSensors Inputs
  // TIM3_CH1, TIM3_CH2, TIM3_CH3
  TIM_SelectHallSensor(TIM4, ENABLE);

  // HallSensor event is delivered with signal TI1F_ED
  // (this is XOR of the three hall sensor lines)
  // Signal TI1F_ED: falling and rising edge of the inputs is used
  TIM_SelectInputTrigger(TIM4, TIM_TS_TI1F_ED);

  // On every TI1F_ED event the counter is resetted and update is tiggered
  TIM_SelectSlaveMode(TIM4, TIM_SlaveMode_Reset);

  // Channel 1 in input capture mode
  // on every TCR edge (build from TI1F_ED which is a HallSensor edge)
  // the timervalue is copied into ccr register and a CCR1 Interrupt
  // TIM_IT_CC1 is fired
    TIM_ICInitTypeDef TIM_ICInitStructure;
  TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;
  TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;
  // listen to T1, the  HallSensorEvent
  TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_TRC;
  // Div:1, every edge
  TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;
  TIM_ICInitStructure.TIM_ICFilter = 0x0;
  TIM_ICInit(TIM4, &TIM_ICInitStructure);


    // channel 2 can be used for commutation delay between hallsensor edge
    // and switching the FET into the next step. if this delay time is
    // over the channel 2 generates the commutation signal to the motor timer
```

```
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
    TIM_OCInitStructure.TIM_Pulse = 0; // 1 is no delay; 2000 = 7ms
    TIM_OC2Init(TIM4, &TIM_OCInitStructure);

     // clear interrupt flag
    //TIM_ClearFlag(TIM4, TIM_FLAG_Trigger);
    //TIM_ClearFlag(TIM4, TIM_FLAG_CC2);
    TIM_ClearFlag(TIM4,TIM_FLAG_CC1);


    TIM_SelectOutputTrigger(TIM4, TIM_TRGOSource_OC2Ref);

    TIM_ITConfig(TIM4, TIM_IT_CC1, ENABLE);
    //TIM_ITConfig(TIM4, TIM_IT_CC2, ENABLE);
    //TIM_ITConfig(TIM4, TIM_IT_Trigger, ENABLE);

  NVIC_InitTypeDef NVIC_InitStructure;
  /* Configure and enable TIM4 interrupt */
  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  TIM_Cmd(TIM4, ENABLE);
}



u8 HALL_Pattern(void){
    return (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_6)<<2 | GPIO_ReadInputDataBit(GPIOB,
GPIO_Pin_7)<<1
        | GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_8)<<0);
}


static const u8 BLDC_BRIDGE_STATE_FORWARD[8][6] =   // Motor step
{
   { 0,0   ,   0,0   ,  0,0 },  //    0 000     000
   { 0,2   ,   1,1   ,  0,1 },  //      4   010     001
   { 1,1   ,   0,1   ,  0,1 },  //      2   001     010
   { 0,1   ,   1,1   ,  0,1 },  //      3   011     011
   { 0,1   ,   0,1   ,  1,1 },  //      6   100     100
   { 0,1   ,   0,1   ,  1,1 },  //      5   110     101
   { 1,1   ,   0,1   ,  0,1 },  //      1   101     110
   { 1,1   ,   1,1   ,  1,1 }   //      0   111     000
};

void BLDCMotorPrepareCommutation(void)
{
    vu8         hallpos;
    vu8         BH1, BL1, BH2, BL2, BH3, BL3;
    const u8    (*bldcBridgeState)[6];
    vs16        pwm;
    pwm = 3600;//(uint16_t) (((uint32_t) 200 * (TimerPeriod)) / 1000);
```

```
    hallpos = HALL_Pattern();


    bldcBridgeState = BLDC_BRIDGE_STATE_FORWARD;


    BH1 = bldcBridgeState[hallpos][0];
    BL1 = bldcBridgeState[hallpos][1];

    BH2 = bldcBridgeState[hallpos][2];
    BL2 = bldcBridgeState[hallpos][3];

    BH3 = bldcBridgeState[hallpos][4];
    BL3 = bldcBridgeState[hallpos][5];


// Bridge FETs for Motor Phase U
    if (BH1) {
        //TIM_SelectOCxM(TIM1, TIM_Channel_1, TIM_OCMode_PWM1);
        //TIM_CCxNCmd(TIM1, TIM_Channel_1, TIM_CCx_Enable);
      //TIM_CCxCmd(TIM1, TIM_Channel_1, TIM_CCx_Disable);
        TIM1->CCR1 = pwm;

    } else {
        TIM1->CCR1 = 0;
        if (BL1){
            //TIM_SelectOCxM(TIM1, TIM_Channel_1, TIM_OCMode_PWM1);
            //TIM_CCxCmd(TIM1, TIM_Channel_1, TIM_CCx_Enable);
        } else {
            //TIM_CCxCmd(TIM1, TIM_Channel_1, TIM_CCx_Disable);
        }
    }

    // Bridge FETs for Motor Phase V
    if (BH2) {
        //TIM_SelectOCxM(TIM1, TIM_Channel_1, TIM_OCMode_PWM1);
        TIM1->CCR2 = pwm;
        //TIM_CCxCmd(TIM1, TIM_Channel_2, TIM_CCx_Enable);
    } else {
        TIM1->CCR2 = 0;
        if (BL2){
            //TIM_CCxCmd(TIM1, TIM_Channel_2, TIM_CCx_Enable);
        } else {
            //TIM_CCxCmd(TIM1, TIM_Channel_2, TIM_CCx_Disable);
        }
    }

    // Bridge FETs for Motor Phase W
    if (BH3) {
        //TIM_SelectOCxM(TIM1, TIM_Channel_1, TIM_OCMode_PWM1);
        TIM1->CCR3 = pwm;
        //TIM_CCxCmd(TIM1, TIM_Channel_3, TIM_CCx_Enable);
    } else {
        TIM1->CCR3 = 0;
        if (BL3){
            //TIM_CCxCmd(TIM1, TIM_Channel_3, TIM_CCx_Enable);
        } else {
            //TIM_CCxCmd(TIM1, TIM_Channel_3, TIM_CCx_Disable);
```

```
        }
    }
}
```

```
void TIM4_IRQHandler(void)
{
    //TIM_ClearITPendingBit (TIM1, TIM_IT_COM);
    //TIM_ClearITPendingBit(TIM4, TIM_IT_Trigger);
    //BLDCMotorPrepareCommutation();
    if (TIM_GetITStatus(TIM4, TIM_IT_CC1) != RESET)
  {
    TIM_ClearITPendingBit(TIM4, TIM_IT_CC1);
   // //commutate ();
    BLDCMotorPrepareCommutation();
      //TIM_GenerateEvent(TIM1, TIM_EventSource_COM);
  }
  else if (TIM_GetITStatus(TIM4, TIM_IT_CC2) != RESET)
  {
    TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
        //commutate ();
    BLDCMotorPrepareCommutation();
  } //else {
    //; // this should not happen
  }
```

best Regards
Pascal