

I'm implementing a BLDC Controller using the STM32F303RE and I've run into an issue where changing direction is causing an over-current caused by a seemingly incorrect output (according to how I've set the CCER, CCMR1 and CCMR2 registers). Wondering if anyone else has run into a similar issue or if anyone has thoughts as to what might be causing it.

Notes and Apologies: This is from a currently in-develop project with lots of un-related code; I'll try to generate a minimal test project using the STM32303E-EVAL dev board in the future, but don't have time to do it right now so I'll only be able to include code snippets and screenshots. I understand that isn't the best way to get help, but appreciate anyone who can take a look / suggest solutions anyway. Thanks.

Overview of Implementation:

- TIM8 is used in 6-step PWM mode to control FET Drivers.
- TIM2 is configured read the Hall signal from the motor to measure velocity and trigger commutation
 - TIM2 is configured to generate an interrupt based on XOR BOTHEDGE change where we read the Hall signals, set the next commutation step and trigger a software commutation event
- OPAMP2, DAC1 Ch2, and COMP6 are configured to give us a hardware Peak Current Detect / BRK behavior

TIM8 Configuration:

```
void periph_bldc_tim_init(void *bldc_tim_handle_void)
{
    GPIO_InitTypeDef gpio_config;

    memset(&gpio_config, 0, sizeof(gpio_config));

    TIM_HandleTypeDef *bldc_tim_handle =
        (TIM_HandleTypeDef *) bldc_tim_handle_void;

    memset(bldc_tim_handle, 0, sizeof(*bldc_tim_handle));

    // Enabled RCC Clk for DAC and Output GPIO
    __HAL_RCC_TIM8_CLK_ENABLE();

    gpio_config.Mode = GPIO_MODE_AF_PP;
    gpio_config.Pull = GPIO_PULLDOWN;
    gpio_config.Speed = GPIO_SPEED_FREQ_HIGH;

    BLDC_1_PWM_TIM_UP_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_UP_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_UP_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_UP_GPIO_PORT, &gpio_config);

    BLDC_1_PWM_TIM_UN_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_UN_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_UN_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_UN_GPIO_PORT, &gpio_config);

    BLDC_1_PWM_TIM_VP_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_VP_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_VP_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_VP_GPIO_PORT, &gpio_config);

    BLDC_1_PWM_TIM_VN_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_VN_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_VN_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_VN_GPIO_PORT, &gpio_config);

    BLDC_1_PWM_TIM_WP_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_WP_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_WP_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_WP_GPIO_PORT, &gpio_config);

    BLDC_1_PWM_TIM_WN_GPIO_CLK_ENABLE();

    gpio_config.Pin = BLDC_1_PWM_TIM_WN_GPIO_PIN;
    gpio_config.Alternate = BLDC_1_PWM_TIM_WN_GPIO_AF;

    HAL_GPIO_Init(BLDC_1_PWM_TIM_WN_GPIO_PORT, &gpio_config);

    HAL_NVIC_SetPriority(TIM8_TRG_COM_IRQn, 2, 0);
    HAL_NVIC_EnableIRQ(TIM8_TRG_COM_IRQn);

    bldc_tim_handle->Instance = TIM8;
    bldc_tim_handle->Init.Prescaler = TIM_CLOCKPRESCALER_DIV1;
    bldc_tim_handle->Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    bldc_tim_handle->Init.Period = (HAL_RCC_GetSysClockFreq()
        * 2) / 20000;
    bldc_tim_handle->Init.CounterMode = TIM_COUNTERMODE_UP;
    bldc_tim_handle->Init.RepetitionCounter = 0;

    if (HAL_OK != HAL_TIM_PWM_Init(bldc_tim_handle))
    {
        DEBUG_SUB_ERROR(DEBUG_BLDC,
            INIT,
            "%s",
            "Error Initializing BLDC 1 PWM TIM\n");
    }

    TIM_SlaveConfigTypeDef bldc_slave_config;

    memset(&bldc_slave_config, 0, sizeof(bldc_slave_config));
}
```

```

// Recently moved from Triggered Commutation to Software Commutation
// Comment out Slave Configuration as we don't use it anymore
//// bldc_slave_config.SlaveMode      = TIM_SLAVEMODE_COMBINED_RESETTRIGGER;
// bldc_slave_config.SlaveMode      = TIM_SLAVEMODE_TRIGGER;
// bldc_slave_config.InputTrigger   = BLDC_1_PWM_TIM_INPUT_TRIGGER;
// bldc_slave_config.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
// bldc_slave_config.TriggerFilter  = 0;
// bldc_slave_config.TriggerPrescaler = TIM_TRIGGERPRESCALER_DIV1;
//
// if (HAL_OK
//     != HAL_TIM_SlaveConfigSynchronization(bldc_tim_handle,
//                                             &bldc_slave_config))
//{
//     DEBUG_SUB_ERROR(DEBUG_BLDC,
//                     INIT,
//                     "%s",
//                     "Error Initializing BLDC 1 PWM_TIM Slave Config\n");
//}

TIM_OC_InitTypeDef pwm_config;

memset(&pwm_config, 0, sizeof(pwm_config));

pwm_config.OCMode      = TIM_OCMODE_PWM1;
pwm_config.OCPolarity   = TIM_OCPOLARITY_HIGH;
pwm_config.OCNPolarity  = TIM_OCNPOLARITY_HIGH;
pwm_config.OCIdleState  = TIM_OCIDLESTATE_RESET;
pwm_config.OCNIdleState = TIM_OCNIDLESTATE_RESET;
pwm_config.OCFastMode   = TIM_OCFAST_DISABLE;

pwm_config.Pulse        = 0;
if (HAL_TIM_PWM_ConfigChannel(bldc_tim_handle, &pwm_config,
    BLDC_1_PWM_TIM_U_TIM_CHANNEL) != HAL_OK)
{
    // Error Handler;
    DEBUG_SUB_ERROR(DEBUG_BLDC,
                    INIT,
                    "%s",
                    "Error During BLDC 1 PWM TIM Channel U Config\n");
}

if (HAL_TIM_PWM_ConfigChannel(bldc_tim_handle, &pwm_config,
    BLDC_1_PWM_TIM_V_TIM_CHANNEL) != HAL_OK)
{
    // Error Handler;
    DEBUG_SUB_ERROR(DEBUG_BLDC,
                    INIT,
                    "%s",
                    "Error During BLDC 1 PWM TIM Channel V Config\n");
}

if (HAL_TIM_PWM_ConfigChannel(bldc_tim_handle, &pwm_config,
    BLDC_1_PWM_TIM_W_TIM_CHANNEL) != HAL_OK)
{
    // Error Handler;
    DEBUG_SUB_ERROR(DEBUG_BLDC,
                    INIT,
                    "%s",
                    "Error During BLDC 1 PWM TIM Channel W Config\n");
}

TIM_BreakDeadTimeConfigTypeDef break_config;

memset(&break_config, 0, sizeof(break_config));

break_config.BreakState      = TIM_BREAK_ENABLE;
break_config.DeadTime         = 20;
break_config.OffStateRunMode  = TIM_OSSR_ENABLE;
break_config.OffstateIDLEMode = TIM_OSSI_ENABLE;
break_config.LockLevel       = TIM_LOCKLEVEL_OFF;
break_config.BreakPolarity    = TIM_BREAKPOLARITY_HIGH;
break_config.BreakFilter     = 0;
break_config.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
break_config.Break2State     = TIM_BREAK2_DISABLE;
break_config.Break2Polarity   = TIM_BREAK2POLARITY_LOW;
break_config.Break2filter    = 0;
if (HAL_TIMEX_ConfigBreakDeadTime(bldc_tim_handle, &break_config)
    != HAL_OK)
{
    // Error Handler
    DEBUG_SUB_ERROR(DEBUG_BLDC,
                    INIT,
                    "%s",
                    "Error During BLDC 1 PWM TIM Break DeadTime Config\n");
}

// Configuration of Commutation Event is needed to enable COM IT
HAL_TIMEx_ConfigCommutationEvent_IT(bldc_tim_handle,
                                     //BLDC_1_PWM_TIM_INPUT_TRIGGER,
                                     //TIM_TS_NONE,
                                     //TIM_COMMUTATION_TRGI);
                                     //TIM_COMMUTATION_SOFTWARE);

// Start timers and disable MOE output
if (HAL_OK != HAL_TIM_Base_Start_IT(bldc_tim_handle))
{
    DEBUG_PRINTF(DEBUG_BLDC,
                 "%s",
                 "Error During PWM 0 Base Start\n");
}

/* ##-3- Start PWM signals generation ##### */
/* Start channel 1 */

```

```
if (HAL_TIM_PWM_Start_IT(bldc_tim_handle, TIM_CHANNEL_1) != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 1 Start\n");
}

/* Start channel 1N */
if (HAL_TIMEX_PWMN_Start_IT(bldc_tim_handle, TIM_CHANNEL_1)
    != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 1N Start\n");
}

/* Start channel 2 */
if (HAL_TIM_PWM_Start_IT(bldc_tim_handle, TIM_CHANNEL_2) != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 2 Start\n");
}

/* Start channel 2N */
if (HAL_TIMEX_PWMN_Start_IT(bldc_tim_handle, TIM_CHANNEL_2)
    != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 2N Start\n");
}

/* Start channel 3 */
if (HAL_TIM_PWM_Start_IT(bldc_tim_handle, TIM_CHANNEL_3) != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 3 Start\n");
}

/* Start channel 3N */
if (HAL_TIMEX_PWMN_Start_IT(bldc_tim_handle, TIM_CHANNEL_3)
    != HAL_OK)
{
    /* Starting Error */

    // Error_Handler();
    DEBUG_PRINTF(DEBUG_BLDC, "%s", "Error During 3N Start\n");
}
```

Overview of Hall / Commutation Table and Direction Control:

Hall Table:

Commutation Table:

```

#define BLDC_U 0
#define BLDC_V 1
#define BLDC_W 2

typedef struct
{
    uint8_t step_num;
    uint8_t high_side;
    uint8_t low_side;
    uint8_t padding;
} bldc_step_t;

const bldc_step_t forward_commutation[MAX_STEP] =
{
    { .step_num = 0, .high_side = BLDC_W, .low_side = BLDC_V },
    { .step_num = 1, .high_side = BLDC_U, .low_side = BLDC_V },
    { .step_num = 2, .high_side = BLDC_U, .low_side = BLDC_W },
    { .step_num = 3, .high_side = BLDC_W, .low_side = BLDC_W }
};

```

Direction Control:

To get the current commutation step we look it up in `forward_commutation` using the current hall_step_num. In the forward direction hall_step_num increases and during reverse the hall_step_num decreases. In either direction, we use the same commutation table ('forward_commutation').

We then use the `.high_side` and `.low_side` settings to set the outputs; how we set the outputs from these settings depends on the direction we're attempting to commutate in.

- For forward direction .high_side will be set to the PWM Highside output and .low_side will be set to Forced Active for Low Side
 - For reverse we swap these to go backwards and .high_side will be set to Forced Active for Low Side and .low_side will be set to the PWM Highside output

```

static void get_comm_step_from_hall(uint8_t motor_num)
{
    int8_t hall_state = bldc_mon_hall_state_get(motor_num);

    bldc_handles[motor_num].last_step = bldc_handles[motor_num].curr_step;
    if((hall_state >=0) && (hall_state < 6))
    {
        bldc_handles[motor_num].curr_step = &forward_commutation[(hall_state) % MAX_STEP];
        if (bldc_handles[motor_num].is_forward)
        {
            bldc_handles[motor_num].wrong_dir_next_step = &forward_commutation[(hall_state + (MAX_STEP - 1)) % MAX_STEP];
            bldc_handles[motor_num].expected_next_step = &forward_commutation[(hall_state + 1) % MAX_STEP];
        }
        else
        {
            bldc_handles[motor_num].wrong_dir_next_step = &forward_commutation[(hall_state + 1) % MAX_STEP];
            bldc_handles[motor_num].expected_next_step = &forward_commutation[(hall_state + (MAX_STEP - 1)) % MAX_STEP];
        }
    }
}

static void update_timer_from_comm_step(bldc_handle_t *handle)
{
    bldc_step_t curr_step;
    curr_step.step_num = handle->curr_step->step_num;

    if(handle->is_forward)
    {
        // Forward, use settings normally
        curr_step.high_side = handle->curr_step->high_side;
        curr_step.low_side = handle->curr_step->low_side;
    }
    else
    {
        // Reverse, swap high and lowside
        curr_step.high_side = handle->curr_step->low_side;
        curr_step.low_side = handle->curr_step->high_side;
    }

    bldc_step_t *step = &curr_step;

    TIM_HandleTypeDef *timer = &handle->timer;

    // Clear all enables and set all modes to PWM Mode 1
    timer->Instance->CCER = 0;
    timer->Instance->CCMR1 &= ~(0x7 << 4) & ~(0x7 << 12); // Clear
    timer->Instance->CCMR1 |= (0x6 << 4) | (0x6 << 12); // PWM Mode 1
    timer->Instance->CCMR2 &= ~(0x7 << 4); // Clear
    timer->Instance->CCMR2 |= (0x6 << 4); // PWM Mode 1

    // Apply current_high side PWM Normal Polarity
    timer->Instance->CCER |= (0x1 << (step->high_side * 4)); // Set High Side High Side for PWM Mode 1

    // Set it to Forced Active (Always on)
    switch (step->low_side)
    {
        // Set mode to Forced Active
        case BLDC_U:
        {
            timer->Instance->CCMR1 &= ~(0x7 << 4); // Clear
            timer->Instance->CCMR1 |= (0x5 << 4); // Forced Active
            break;
        }
        case BLDC_V:
        {
            timer->Instance->CCMR1 &= ~(0x7 << 12); // Clear
            timer->Instance->CCMR1 |= (0x5 << 12); // Forced Active
            break;
        }
        case BLDC_W:
        {
            timer->Instance->CCMR2 &= ~(0x7 << 4); // Clear
            timer->Instance->CCMR2 |= (0x5 << 4); // Forced Active
            break;
        }
        default:
        {
        }
    }

    // Set new low_side to enabled and non-inverted (forced active)
    timer->Instance->CCER |= (0x1 << ((step->low_side * 4) + 2));

    // Need to Force Inactive the channel for which neither side is being driven
    // Otherwise it goes high-impedance
    uint8_t unused = 0x3;
}

```

```

unused &= ~(step->low_side | step->high_side);

// First set mode to Force Inactive
switch (unused)
{
    // Set mode to Forced InActive
    case BLDC_U:
    {
        timer->Instance->CCMR1 &= ~(0x7 << 4); // Clear
        timer->Instance->CCMR1 |= (0x4 << 4); // Forced In-Active
        break;
    }
    case BLDC_V:
    {
        timer->Instance->CCMR1 &= ~(0x7 << 12); // Clear
        timer->Instance->CCMR1 |= (0x4 << 12); // Forced In-Active
        break;
    }
    case BLDC_W:
    {
        timer->Instance->CCMR2 &= ~(0x7 << 4); // Clear
        timer->Instance->CCMR2 |= (0x4 << 4); // Forced In-Active
        break;
    }
    default:
    {
    }
}

// Set new low_side to enable non-inverted (forced in-active)
timer->Instance->CCER |= (0x1 << ((unused * 4) + 2));

```

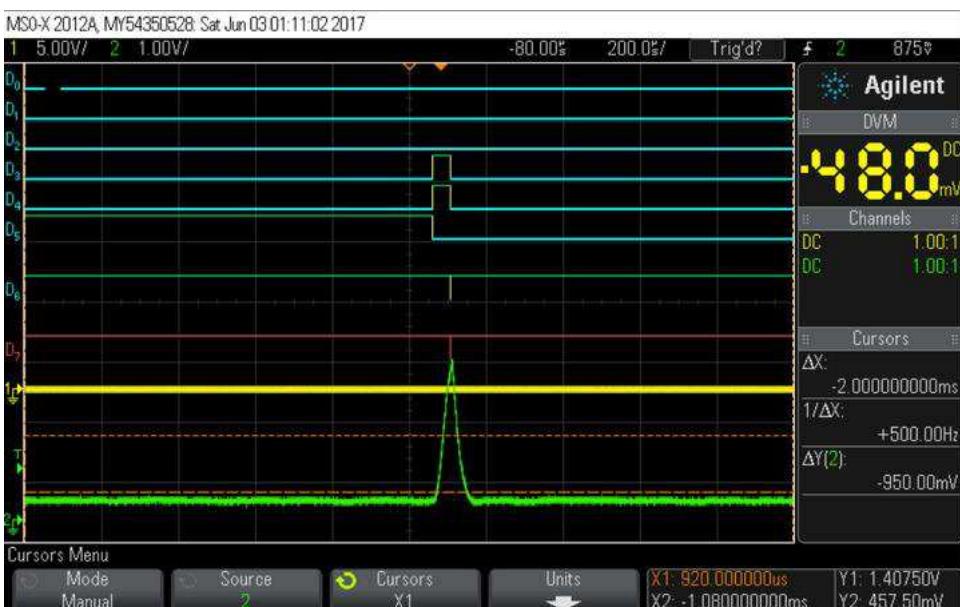
Statement of the Problem:

To Change direction I take the current output (e.g., BLDC_U is the high-side set to PWM Mode 1 and BLDC_V is set to low_side Forced Active) and invert it (e.g., BLDC_U is the low_side set to Forced Active and BLDC_V is set to high_side PWM Mode 1). I only change direction when the motor is stopped so all CCRx registers are set to 0 and the output of PWM Mode 1 should be 0 (as CNT > CCRx always).

- I should see the Low-side of BLDC_V go from Forced Active to Forced Inactive and the High-Side of BLDC_V go from Forced Inactive to PWM Mode 1 Inactive.
 - Instead, I see BLDC_V High-side go immediately active and cause an over-current
 - Low-Side works as expected
 - All channels exhibit this behavior on High-Side when making this transition
 - This transition does not occur during the normal commutation pattern and thus I don't see this issue during normal commutation
 - Seems like a commutation where we swap CCxNE and CCxE (and likely also change OCxM) isn't taking and leaving BLDC_x High-Side in Forced Active until over-current BRK disable outputs
 - Forcing additional Commutations doesn't appear to resolve the issue, though manually turning motor and trigger Hall commutation does

Image of the issue:

- D0 = BLDC_U High-side
 - D1 = BLDC_U Low-side
 - D2 = BLDC_V High-Side
 - D3 = BLDC_V Low-Side
 - D4 = BLDC_W High-Side
 - D5 = BLDC_W Low-Side
 - D6 = Hall U
 - D7 = Hall V
 - 1(Yellow) = Hall W
 - 2(Green) = Combined Motor Current



Debug Logs that indicate settings of CCER, CCMR1, CCMR2, MOE, OSSR, and OSSI aren't changing:

- H1:Hall is Hall Interrupt Event
 - Cam_Dir+ and output of CNT etc is Pre commutation setup
 - COM is actual COM event that should move Com_Dir setup from Pre-load to actual registers

- Note: This log isn't from the same run as the above screenshot.

Drive:0, Prev_Dir: 1, Dir: 1, duty:0, CCR1:0, CCR2:0, CCR3:0
H1:Hall:4
Com_Dir+
CNT: 1489, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000414, CCMR1:0x00006858, CCMR2:0x00000048
COM

H1:Hall:5
Com_Dir+
CNT: 2021, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000144, CCMR1:0x00004858, CCMR2:0x00000068
COM

H1:Hall:0
Com_Dir+
CNT: 4910, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000144, CCMR1:0x00005848, CCMR2:0x00000068
COM

H1:Hall:1
Com_Dir+
CNT: 5796, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000441, CCMR1:0x00005868, CCMR2:0x00000048
COM

H1:Hall:2
Com_Dir+
CNT: 5148, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000441, CCMR1:0x00004868, CCMR2:0x00000058
COM

H1:Hall:3
Com_Dir+
CNT: 3164, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000414, CCMR1:0x00006848, CCMR2:0x00000058
COM

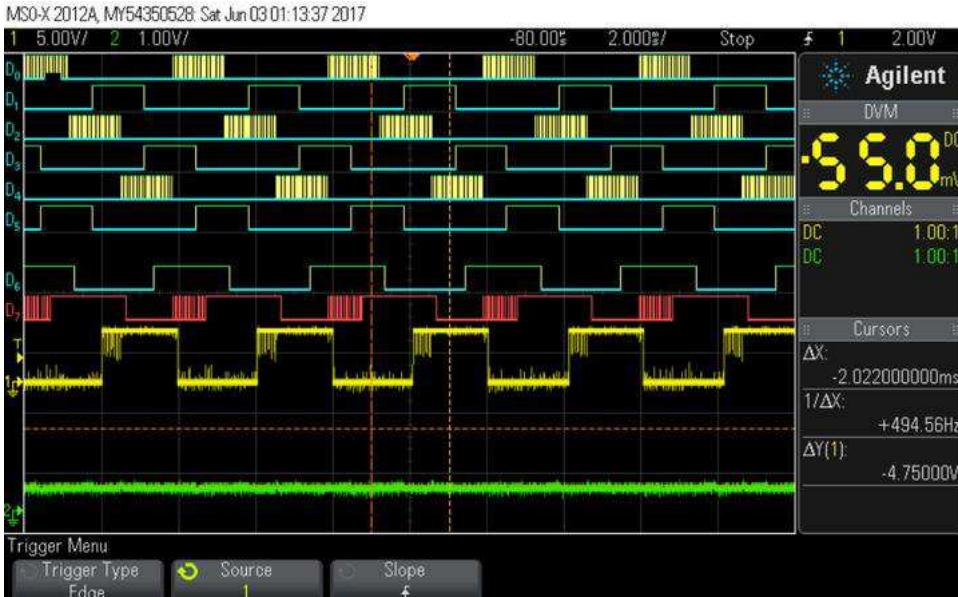
H1:Hall:4
Com_Dir+
CNT: 1438, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000414, CCMR1:0x00006858, CCMR2:0x00000048
COM

Drive:0, Prev_Dir: 1, Dir: 1, duty:0, CCR1:0, CCR2:0, CCR3:0
...Currently Stopped,
...Later When we try to change Direct
...
Drive:0, Prev_Dir: 1, Dir: 1, duty:0, CCR1:0, CCR2:0, CCR3:0

```
COM_D
CNT: 6293, MOE:32768, OSSR:2048, OSSI:1024
CCER:0x00000441, CCMR1:0x00005868, CCMR2:0x00000048
COM
BLDC 1 COMP Interrupt
BLDC 1 COMP Interrupt
Drive:0, Prev_Dir: 1, Dir: 0, duty:72, CCR1:0, CCR2:0, CC
Motor Fault: 4, Motor Num: 1
Stopping BLDC 1
Free Wheel: 1
-
CNT: 106, MOE:0, OSSR:2048, OSSI:1024
CCER:0x00000441, CCMR1:0x00005868, CCMR2:0x00000048
COM
Drive:0, Prev_Dir: 0, Dir: 0, duty:0, CCR1:72, CCR2:72, C
```

Image of Normal Commutation (Same Signals):

- Noise on Hall Signals is filtered out before MCU (Just not easy to get at those signals)



Current Workaround:

It seems that I can avoid this issue if instead of swapping from CCxE to CCxNE in a single commutation cycle if I first disable CCxE, Commutate (so that outputs are no longer driven by timer (MOE=1, CCxE=0, CCxNE=0)), set CCxE and Commutate this issue doesn't occur.

I am not sure why first going through Output Disabled state changes the behavior.

The above does not actually appear to resolve this issue.