

1) **setParameter(HALL\_POLE\_PAIRS,2)**

Used for speed calculation from commutation period.

```
* speed [rpm] = numerator / (commutation period [256/fSYS in Hz])
* numerator = CCU6 timer frequency [Hertz] * 60 / (6 * pole pairs),
round to nearest */
```

```
//#define SPEED_FROM_COMM_PERIOD_NUM ((uint32)((((EMO_FSYS_HZ / 256.0)
* 60.0) / (6.0 * (float)BCHALL_POLE_PAIRS)) + 0.5))
```

2) **setParameter(HALL\_PWM\_FREQ,20000, BOARD1)**

```
ccu6_t12_freq = Emo_Hallpar_Cfg.PWM_Frequency;
```

The timer T12 block is the main unit to generate the 3-phase PWM signals.

3) **setParameter(HALL\_INIT\_DUTY,30, BOARD1)**

Set common duty cycle.

```
InitDutyCycle = (uint16)((((uint32)Emo_Hallpar_Cfg.initDutyCycle) *
ccu6_t12pr)/100);
```

Load compare value to the shadow register

```
CCU6_LoadShadowRegister_CC60(InitDutyCycle);
```

For the Hall and Modulation output patterns, a double-register structure is implemented.

While register MCMOUT holds the actually used values, its shadow register MCMOUTS can be loaded by software from a predefined table, holding the appropriate Hall and Modulation patterns for the given motor control. A transfer from the shadow register into register MCMOUT can take place when a correct Hall pattern change is detected.

4) **setParameter(HALL\_OFFSET\_60DEGREE\_EN,1,BOARD1)**

```
#define Ccu6_SetPtns(CURHS, EXPHS, MCMPS) \
( (((uint32)(MCMPS)) << 0u) | \
((uint32)(EXPHS)) << 8u) | \
((uint32)(CURHS)) << 11u ) )

void init_emoccucfg(uint8 offset60deg_en)
{
    if(offset60deg_en == 0)
    {
        EmoCcu_Cfg.HallOutPtns[0] = (uint16)Ccu6_SetPtns(0, 0, 0x00); /* Hall pattern=0, forward direction (error) */
        EmoCcu_Cfg.HallOutPtns[1] = (uint16)Ccu6_SetPtns(1, 3, 0x31); /* Hall pattern=1, forward direction */
        EmoCcu_Cfg.HallOutPtns[2] = (uint16)Ccu6_SetPtns(2, 6, 0x07); /* Hall pattern=2, forward direction */
        EmoCcu_Cfg.HallOutPtns[3] = (uint16)Ccu6_SetPtns(3, 2, 0x34); /* Hall pattern=3, forward direction */
        EmoCcu_Cfg.HallOutPtns[4] = (uint16)Ccu6_SetPtns(4, 5, 0x1C); /* Hall pattern=4, forward direction */
        EmoCcu_Cfg.HallOutPtns[5] = (uint16)Ccu6_SetPtns(5, 1, 0x0D); /* Hall pattern=5, forward direction */
        EmoCcu_Cfg.HallOutPtns[6] = (uint16)Ccu6_SetPtns(6, 4, 0x13); /* Hall pattern=6, forward direction */
        EmoCcu_Cfg.HallOutPtns[7] = (uint16)Ccu6_SetPtns(0, 0, 0x00); /* Hall pattern=7, forward direction (error) */
        EmoCcu_Cfg.HallOutPtns[8] = (uint16)Ccu6_SetPtns(0, 0, 0x00); /* Hall pattern=0, reverse direction (error) */
```

If its enable then the current, expected and corresponding output pattern is changed.

5) **setParameter(HALL\_ANGLE\_DELAY\_EN,0,BOARD1) and**

**setParameter(HALL\_DELAY\_ANGLE,0,BOARD1)**

```
/* Set T13 period to Hall delay time, limit to minimum = Hall filter
time */
```

```
DelayTime = (uint16)(((((uint32)DiffTime) *
EmoCcu_HallStatus.DelayAngle) + 30u) / 60u);
if(DelayTime < T13_HALL_FILTER_TIME_TICKS)
{
    DelayTime = T13_HALL_FILTER_TIME_TICKS;}
```

- 6) `setParameter(HALL_DELAY_MINSPEED,10,BOARD1)`  
Minimum (absolute) speed for Hall delay
- 7) `setParameter(HALL_SPEED_KP,40, BOARD1)` and  
`setParameter(HALL_SPEED_KI,20, BOARD1)`  
Set KP and Ki for speed loop
- 8) `speedPiMax; speedPiMin; speedIMax; speedIMin;`  
Used to put saturation limit for the integrator and pi loop.