# UM1594
# User manual

Six-step sensorless BLDC motor drive example software for the STM32™

## Introduction

The example program is written specifically to run on the STEVAL-IHM040V1 board using the STM32F100C8T6 but can be adapted to run on other STM32 microcontrollers with minor modifications. The program is written in C and provides the main functions of sensorless synchronization to the permanent magnet rotor position of a 3-phase BLDC motor along with 3-phase bridge commutation, speed regulation, acceleration/deceleration control, and gating control logic which is specific to the STMicroelectronics® SLLIMM™ module STGIPN3H60. The driving logic can be easily adapted to any monolithic or discrete 3-phase bridge which requires six logic control signals. Deadtime functionality is provided within the STM32.

# Contents

# 1 Theory of operation

The structure of the program consists of a continuously running background loop with one time driven interrupt. Pulse width modulation for the motor control and time base interrupt are both implemented using the advance timer (Timer1) in the STM32. The time driven interrupt is triggered once each period by Timer1 channel 4. The signal from channel 4 triggers the execution of the “PWM Interrupt Service Routine” (PWMISR) but is not seen on any hardware output from the processor. PWMISR executes once each PWM cycle just before the end of the cycle (during the “off” period, just before the start of the next cycle when the transistors are turned back on). Having CH4 available on Timer1 along with the 3 PWM channels, which actually control the transistor bridge, gives the ability to precisely place the synchronous execution of the PWMISR anywhere within the PWM period. The primary function of the PWMISR is to sample the motor terminal voltage of the floating phase in order to capture line to neutral back EMF (BEMF). This information is processed by the STARTSTATE state machine (also in the PWMISR) in order to determine BEMF zero crossings and, ultimately, bridge commutation timing. The PWMISR also handles startup alignment and open loop ramp-up as well as motor current sampling. Since the PWM period is fixed at 50 microseconds (20 KHz), the interrupt runs once every 50 microseconds and serves as the timing heartbeat for the entire program.

The program also includes one event driven interrupt, the Timer1 break interrupt. Execution of this interrupt service routine is triggered by a falling edge on the Timer1 break input pin, which is a hardware indication of an uncontrolled overcurrent situation. This interrupt service routine does nothing more than set the global overcurrent flag true. The outputs that drive the inverter bridge are automatically set to their safe state via hardware within the STM32.

The background loop consists of non-critical code that runs at an uncontrolled rate (once per loop) along with execution rate controlled sections that run at 100 microseconds, 1 millisecond, and 10 millisecond intervals. Execution of these routines is not directly driven by a timer interrupt but is, instead, driven by the polling of individual software timers that are updated by the PWMISR. In this way, we are assured that these routines will run, on average, 10K, 1K, and 100 times per second but they do not present any interrupt priority management issues.

The 100 microsecond routine has only one function, which is to implement a state variable type observer which accepts the motor commutation step as its input (as determined by the STARTSTATE state machine) and outputs an actual relative motor position, an observed motor position, and an observed speed. The observed speed is used as feedback for the speed regulator.

The 1 millisecond routine executes the timed overload function and the speed regulator. The overload functions as a timed “pull back” type of controller which will decrease an overall limit on applied duty cycle when the motor current exceeds the overload current threshold for more than the allowable time. This allows the bridge to operate at higher current levels for short time durations to allow for transient loads and motor acceleration, but prevents operation at these levels for long periods that would eventually cause an overtemperature condition on the SLLIMM module. The speed regulator monitors the speed reference and speed estimate to determine a speed error and, from this, generates the commanded duty cycle needed to keep the motor running at the desired speed. The speed regulator is a conventional proportional plus integral type algorithm.

The 10 millisecond routine scales the speed command potentiometer (which is actually sampled in the PWMISR) and also executes the acceleration/deceleration control algorithm which translates the speed command (as received from the command potentiometer) to the slew rate limited speed reference.

The free running section of the background loop manages the flashing LED display to indicate fault status. The controlled execution rate sections of the background loop (100 microsecond, 1 millisecond, and 10 millisecond) are triggered by use of the variables heartbeat1time, heartbeat2time, heartbeat3time, and globalcounter. Globalcounter is a 32-bit unsigned software timer which is incremented with each run of the PWMISR. Since the PWMISR is timer controlled to run once each 50 microseconds, each “tick” of the globalcounter is “worth” 50 microseconds. The globalcounter is continually incremented and when it reaches the limit of its range, it will naturally roll over to zero and continue. Each time one of the timed sections runs, it will “grab” the current value of globalcounter and save it in its heartbeat (1, 2, or 3) time. Later, this heartbeat time (the time of the last execution) is subtracted from the current time to calculate the time elapsed since the last execution. When this elapsed time matches or exceeds the run period for a particular section, the routine is executed. The 2's compliment arithmetic naturally handles any possible problems arising from the roll over of globalcounter.

Each of the blocks of code is described by pseudocode in the following sections. Please note that the double slash indicates that the remainder of the text line is a clarifying comment for the reader.

# 2 PWMISR (PWM interrupt service routine)

Convert BEMF of floating phase via ADC and store in variable bemfsample

Increment zccounter, alignmentcounter, holdcounter, and ledtime while ADC is converting (timers)

```
If (autostep flag is true) // ramping up or holding speed
{
Increment commcounter // measures time between steps in units of PWM periods
   If (commcounter>step)
   {
   Commcounter = 0
   Increment phase (rolling over to zero after five)
   Increment position
   Lookup TIM1_CCER value from table based on phase to commutate bridge
   }
}
If (run flag is false) startstate = 0
Execute the startstate state machine
Determine and store ADC channel of BEMF to convert on next PWMISR run //
based on step
Convert motor current signal via ADC (with bipolar offset) and store in
variable ifb
Increment globalcounter while ADC is converting
Ifbsum = ifbsum + ifb // to be used in average current calculation
Increment ifbcount // to be used in average current calculation
Clear hardware interrupt
```

# 3 Startstate state machine

```
Case 0: // motor is stopped
Turn off bridge
If (run flag is true)
{
Execute motorstartinit function to initialize variables
Advance case to 5
}
Case 5: //set up alignment
Set duty cycle per alignmentdc
Phase = 0
Lookup TIM1_CCER value from table based on phase to commutate bridge
Alignmentcounter = 0
Advance case to 10

Case 10: //timing out alignment
If (alignmentcounter>alignmenttime)
{
Rampspeed = 1
Commcounter = 0
Autostep flag = true
Set dutycycle per rampupdc
Advance case to 20
}
Case 20: // ramping up
Rampspeed = rampspeed + ramprate
Long0 = 4,000,000,000 / rampspeed // convert speed to step period
If (long0>30,000) long0 = 30,000 // limit step time to prevent math rollover
Step = long0
If (step<minstep) // check for end of speed ramp
{
Holdcounter = 0
Advance case to 100
}
Case 100: // wait at constant speed for hold time
If (holdcounter>holdtime) advance case to 110

Case 110: // wait until we are in step 5
If (phase = 5) advance case to 120

Case 120: // wait for leading edge of step 0 (commutation)
If (phase = 0)
{
```

```
Demagcounter = 0
Demagthreshold = step * demagallowance/256
Advance case to 130
}
Case 130: // wait out demag time
Increment demagcounter
If (demagcounter>demagthreshold) advance case to 140

Case 140: // looking for zero crossing of BEMF
If (risingedge flag is true) // if we are looking for a rising edge of BEMF
{
   If (bemfsample>zcthreshold)
   {
   If (zcfound flag is true) step = zccounter
   Commthreshold = step * risingdelay/256
   Zccounter = 0
   Commcounter = 0
   Set risingedge flag to false
   Set zcfound flag to true
   Set autostep flag to false
   Advance case to 150
   }
}
Else // looking for falling edge of BEMF
{
   If (bemfsample<zcthreshold)
   {
   If (zcfound flag is true) step = zccounter
   Commthreshold = step * fallingdelay/256
   Zccounter = 0
   Commcounter = 0
   Set risingedge flag to true
   Set zcfound flag to true
   Set autostep flag to false
   Advance case to 150
   }
}

Case 150: // wait out commutation delay (from zero crossing)
increment commcounter
If (commcounter>commthreshold) // commutate
{
increment position // used by speed observer
Increment phase (rolling over to zero after five)
Lookup TIM1_CCER value from table based on phase to commutate bridge
```

```
demagcounter = 0
demagthreshold = step * demagallowance/256
set state back to case 130 to wait out demag
   If (phase = 0) // calculate current average over one cycle
   {
   ifbave = (ifbsum*4)/ifbcount // average current scaled for use by
overload routine
   ifbcount = 0
   ifbsum = 0
   }
}

// end of startstate state machine
```

# 4 100 microsecond routine

```
heartbeat1time = globalcounter
slong0 = position - (positionest/4096) // position observer error
(positionest scaled 4096X to improve resolution)
speedest = slong0
potitionest = positionest + speedest // integrate speedest to get
positionest
// check for and prevent eventual math rollover
If (positionest & 0x80000000) // if positionest is greater than ½ full scale
for variable size
{
positionest = positionest & 0x7FFFFFFF // subtract ½ full scale to prevent
eventual math rollover
position = position - 524288 // subtract ½ full scale / 4096 (to account for
scaling difference)
}
//************** end of 100 microsecond routine ***************
```

# 5 1 millisecond routine

```
heartbeat2time = globalcounter
// overlaod function
If (ifbave>overloadthreshold AND overload flag is false) overloadcounter =
overloadcounter + overloaduprate
else overloadcounter = overloadcounter - overloaddownrate
If (overloadcounter<0)
{
overloadcounter = 0
overloadflag = 0
}
If (overloadcounter>1,000,000) set overloadflag true
If (ifb>overloadsetpoint) AND overloadflag is true) overloaddclimit =
overloaddclimit-3
else overloaddclimit = everloaddclimit + 1
If (overloaddclimit>1000) overloaddclimit = 1000;
If (overloaddclimit<100) overloaddclimit = 100;

// *************** speed regulator *********************************
long0 = (speedest*6250)/256// calculate rpm from speedest and number of
pole pairs
rpm = long0/polepairs
slong0 = rpmref - rpm; // speed error
propterm = (slong0*propgain)/256;// calculate proportional term of
proportional plus integral
errorint = errorint + slong0; // integrate speed error to get raw integral
term

// set activedc to the lower of maxdc or overloaddclimit
If (maxdc<overloaddclimit) activedclimit = maxdc;
else activedclimit = overloaddclimit;

// limit integral term so that P + I will be less than the active duty cycle
limit
slong0 = activedclimit - propterm;
If (slong0<0) slong0 = 0; // slong0 holds maximum allowable intterm
slong0 = slong0 * intclampscaler; // slong0 holds max error integral
If (errorint>slong0) errorint = slong0;
If (errorint<0) errorint = 0;
intterm = (errorint*intgain)>>10; // apply integral gain
slong0 = propterm + intterm; // combine proportional and integral
If (slong0>activedclimit) slong0 = activedclimit;
If (slong0<100) slong0 = 100;
```

```
// count out delay after rotor is synced to enable speed regulator
If ((zcfound) && (transitioncounter<100)) transitioncounter++;
If (transitioncounter<100)
{
runningdc = 500;
errorint = 500 * intclampscaler;
rpmref = rpm;
}
else
{
runningdc = slong0;
}

If (zcfound)
{
TIM1->CCR1= runningdc;
TIM1->CCR2 = runningdc;
TIM1->CCR3 = runningdc;
}
// ************** end of 1 millisecond routine *************************
```

# 6 10 millisecond routine

```
heartbeat3time = globalcounter;
potvalue = 4095 - adcread(2); // read pot channel
If (potvalue>200) set run flag true;
If (potvalue<100) set run flag false;
rpmcmd = potvalue*4;
If (rpmcmd<100) rpmcmd = 100;
// accel/decel control
slong0 = rpmcmd-rpmref;
If (slong0>acclim) slong0 = acclim;
If (slong0<-declim) slong0 = -declim;
rpmref = rpmref + slong0;
 // ************** end of 10 millisecond routine *************************
```

## Background code that runs every loop

```
If (overcurrent flag is true) flashcount = 2;
execute ledstate state machine
```

# 7 LEDstate state machine

```
// NOTE: if flash count is zero, the LED will stay on continuously.
Otherwise, the LED will flash on and off
//the number of times dictated by flashcount, with a long pause in between
to allow counting the flashes
Case 0:
If (flashcount = 0) turn LED on // normal state, no fault, LED on steady
else
{
turn LED off
ledtime = 0;
advance to case 10
}
Case 10: // waiting out long pause
If (ledtime>30000)
{
flashcounter = 0;
advance to case 20
}
Case 20: // long pause over
ledon;
ledtime = 0;
advance to case 30
Case 30: // waiting out short on time
If (ledtime>8000)
{
ledoff;
ledtime = 0;
advance to case 40
}
Case 40: // waiting out short off time
If (ledtime>8000)
{
flashcounter++;
if (flashcounter> = flashcount) go back to case 0
else go back to state 20

 // ***************end of ledstate switch statement************
```

# 8 Variable descriptions

**Table 1. Variable descriptions**

| Variable | Type | Description |
|---|---|---|
| overloadflag | 8-bit logic flag | If true, overload conditions exists and current is limited. |
| overcurrent | 8-bit logic flag | If true, overcurrent shutdown has occurred. |
| zcfound | 8-bit logic flag | If true, the first BEMF zero crossing has been found, rotor sync. has been established. |
| autostep | 8-bit logic flag | If true, system is either in ramping up or hold speed stepping mode. |
| run | 8-bit logic flag | If true, drive is either starting up or running. |
| risingedge | 8-bit logic flag | If true, looking for BEMF rising edge next, else looking for falling edge. |
| flashcounter | 8-bit unsigned | Used by the ledstate state machine to keep track of the number of LED flashes. |
| phase | 8-bit unsigned | Takes on the values from 0 to 5 to indicate the current commutation step. |
| risingdelay | 8-bit unsigned | Controls the delay from BEMF zero crossing detection (rising edge) until the bridge is commutated. 0 to 255 represents 0 to 255/256 of a step (60 electrical degrees). The nominal theoretical value is 128, which is 30 electrical degrees. This is the phase shift between line to neutral ZC (which is sensed) and line to line ZC, proper commutation time. Reducing will advance commutation timing. |
| fallingdelay | 8-bit unsigned | Controls the delay from BEMF zero crossing detection (falling edge) until the bridge is commutated. 0 to 255 represents 0 to 255/256 of a step (60 electrical degrees). The nominal theoretical value is 128, which is 30 electrical degrees. This is the phase shift between line to neutral ZC (which is sensed) and line to line ZC, proper commutation time. Reducing will advance commutation timing. |
| ledstate | 8-bit unsigned | State variable for the ledstate state machine. |
| flashcount | 8-bit unsigned | Set to command the required number of LED flashes to be displayed. |
| startstate | 8-bit unsigned | State variable for startstate state machine. |
| bemfchannel | 8-bit unsigned | Predetermined ADC channel of next BEMF (phase A, B, or C) to be sampled. |
| maxdc | 16-bit unsigned | 0 to 1200 represents 0 to 100% duty cycle. User designated max used to clamp the output of the speed regulator. This is superseded by a lower value in case of overload. |
| overloaddclimit | 16-bit unsigned | 0 to 1200 represents 0 to 100%. This is controlled by the overload function and will be brought below the maxdc setting in case of overload, where it will supersede. |
| activedclimit | 16-bit unsigned | 0 to 1200 represents 0 to 100%. This will always be the lesser of maxdc or overloaddclimit and is always used as the actual limit. |
| ifboffset | 16 -bit unsigned | 0 to 4095 (ADC counts) represents 0 to 3.3 V DC. The current sensing signal is read 1024 times at startup and averaged to determine this value. Since current is known to be zero, this represents the zero current offset of the signal. During normal operation, this value is subtracted from each raw current conversion to correct for the offset. |
| commcounter | 16-bit unsigned | Used to measure the time delay from zero crossing to commutation. Each count represents one PWM period or 50 microseconds. |

Table 1. Variable descriptions (continued)

| Variable | Type | Description |
|---|---|---|
| step | 16-bit unsigned | Holds the last measured zero crossing to zero crossing interval. Each count represents one PWM period or 50 microseconds. |
| bemfsample | 16-bit unsigned | 0 to 4095 (ADC counts) represents 0 to 3.3 V DC. This is the most recent reading of motor terminal voltage from the floating phase. Scaling is 1 to 1 but the value is clamped in hardware since we are only looking for zero crossings (positive or negative). |
| demagcounter | 16-bit unsigned | Used to measure the demag time allowance, which is the required waiting time from commutation until valid BEMF sampling can resume. This allows time for the current in the floating phase to fall to zero. Each count represents one PWM period or 50 microseconds. |
| zccounter | 16-bit unsigned | Used to measure the time between zero crossings . Each count represents one PWM period or 50 microseconds. zccounter value is incremented each PWM cycle. Its value is transferred to step and it is cleared when ZC is detected. |
| potvalue | 16-bit unsigned | Most recent reading of command potentiometer. 0 to 4095 represents 0 to 100%. |
| runningdc | 16-bit unsigned | 0 to 1200 represents 0 to 100%. This is the actual duty cycle being applied to the bridge when the motor is running. |
| commthreshold | 16-bit unsigned | Sets the time interval between zero crossing and commutation. Each count represents one PWM period or 50 microseconds. |
| demagthreshold | 16-bit unsigned | Sets the time interval between commutation and the resumption of BEMF sampling. Each count represents one PWM period or 50 microseconds. |
| transitioncounter | 32-bit unsigned | Each “tick” is 1 millisecond. This software timer is used to measure out an interval of 100 milliseconds after rotor sync. is achieved and before the speed regulator is enabled, when the duty cycle is held constant to allow the system to stabilize. |
| position | 32-bit unsigned | Each count is one step or 60 electrical degrees. This variable holds a relative rotor position. It is incremented with each motor commutation but it is occasionally corrected so that it will not overflow. It serves as the primary input to the speed observer. |
| positionest | 32-bit unsigned | This is a state variable of the speed observer. In the steady state, it will follow position but is scaled to be 4096 times bigger to enhance resolution. |
| heartbeat1time | 32-bit unsigned | Holds the value of globalcounter at the last execution of the 100 microsecond routine. Each tick is one PWM cycle. It is used to control the execution rate of the routine. |
| heartbeat2time | 32-bit unsigned | Holds the value of globalcounter at the last execution of the 1 millisecond routine. Each tick is one PWM cycle. It is used to control the execution rate of the routine. |
| heartbeat3time | 32-bit unsigned | Holds the value of globalcounter at the last execution of the 10 millisecond routine. Each tick is one PWM cycle. It is used to control the execution rate of the routine. |
| globalcounter | 32-bit unsigned | This is a global software timer which is incremented with each run of the PWM interrupt service routine. It is a free running timer which is allowed to naturally roll over. |

Table 1. Variable descriptions (continued)

| Variable | Type | Description |
|---|---|---|
| holdcounter | 32-bit unsigned | This is a software timer that is used to measure the time that the motor is held at a constant speed (in open loop stepping mode) at the end of the ramp-up period, before BEMF sampling is commenced. Each “tick” in 50 microseconds. |
| alignmentcounter | 32-bit unsigned | This is a software timer that is used to measure the time that the motor is held at the alignment stage before the ramp-up period. Each “tick” in 50 microseconds. |
| rampspeed | 32-bit unsigned | This variable controls the stepping rate during ramp-up. It increases in value during ramp-up as the stepping rate is increased. The constant value of 4,000,000,000 is divided by rampspeed to get the current step time (in units of PWM cycles) |
| ledtime | 32-bit unsigned | Used by the ledstate state machine to time out various time intervals. Each “tick” is one PWM cycle. |
| speedest | 16-bit signed | Raw speed estimate output from speed observer. |
| rpm | 16-bit signed | Scaled speed derived from speedest and polepairs. Unit is revolutions per minute. |
| rpmcmd | 16-bit signed | Raw, scaled RPM command. This is potvalue*4 so the range is 0 to 16,380. Unit is revolutions per minute. |
| rpmref | 16-bit signed | Follows rpm cmd with a slew limit imposed. |
| ifbcount | 16-bit signed | Used to count the number of IFB samples summed into ifbsum. This is used during the calculation of ifbave. |
| ifbave | 16-bit signed | Average motor current over one full electrical cycle. Used in overload determination. |
| ifb | 16-bit signed | Most recent motor current sample. |
| errorint | 32-bit signed | Raw time integral of speed regulator error. This value is dynamically clamped so that the proportional plus integral sum does not exceed the duty cycle limit. |
| ifbsum | 32-bit signed | Summation of current samples taken in a given electrical cycle. It is used to calculate the average. |
| overloadcounter | 32-bit signed | This variable is incremented and decremented in a weighted manner depending on whether average current is above or below the overload threshold. The value is clamped so that it cannot go below zero. When it exceeds a fixed threshold, overload mode is initiated. |

# 9 Compile time settings (#defines)

Table 2. Compile time settings (#defines)

| Setting | Type | Description |
|---|---|---|
| propgain | Numeric | Speed regulator proportional gain. |
| intgain | Numeric | Speed regulator integral gain. |
| acclim | Numeric | Acceleration limit. |
| declim | Numeric | Deceleration limit. |
| polepairs | Numeric | Number of motor pole pairs. Total number of poles/2. |
| alignmentdc | Numeric | 0 to 1200 represents 0 to 100% duty cycle. This is the duty cycle applied to the bridge during alignment. |
| rampupdc | Numeric | 0 to 1200 represents 0 to 100% duty cycle. This is the duty cycle applied to the bridge during the ramp-up and hold portions of the starting sequence. |
| zcthreshold | Numeric | 0 to 4095 represents 0 to 3.3 V DC. This is the reference value that the BEMF sample is compared against to determine zero crossings. This should normally be set very low but may be set higher with high BEMF motors to improve noise immunity. |
| alignmenttime | Numeric | Units are PWM cycles (50 microseconds). This parameter sets the time that the motor start-up sequence spends in rotor alignment. This should be set long enough that the rotor comes to a stop before ramp-up commences. The required time will be influenced by rotor + load inertia and system mechanical damping. |
| demagallowance | Numeric | 0 to 255 represents 0 to 255/256 of a step time. This sets the time that the state machine will wait after commutation before beginning to sample BEMF. |
| holdrpm | Numeric | Units are revolutions per minute. This sets the motor speed at the end of the start-up ramp. |
| holdtime | Numeric | Units are PWM cycles (50 microseconds). This parameter sets the time that the motor is held (in stepping mode) at hold rpm before rotor sync. is started. |
| startuprpmpersecond | Numeric | Controls acceleration during ramp-up. |
| overloadseconds | Numeric | Sets the number of seconds that motor current is permitted to stay above the overload threshold before overload mode is entered. In overload mode, current is “pulled back” to the overload threshold level. |
| overloadsecondsreset | Numeric | Sets the number of seconds that current must stay below the overload threshold before overload mode is cancelled. |
| continuouscurrent | Numeric | Units are milliamperes. This is the continuous current rating used by the overload function. Extended operation above this level will activate the overload mode. |

# 10 Revision history

Table 3. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 14-Jan-2013 | 1 | Initial release. |