Implementing Embedded Speed Control for Brushless DC Motors
Part 2

Yashvant Jani
Renesas Technology America, Inc.
450 Holger way, San Jose CA 95134
408-382-7716
yashvant.jani@renesas.com

Abstract
Brushless Direct Current (BLDC) motors, also known as permanent magnet motors, are used today in many applications. A new generation of microcontrollers and advanced electronics has overcome the challenge of implementing required control functions, making BLDC motors more practical for a wide range of uses.

This two-part seminar covers BLDC motor control fundamentals and implementation techniques. Part 1 discusses 120-degree trapezoidal control with and without sensors, while Part 2 covers 180-degree sine wave modulation and V/f open-loop and closed-loop control with sensors. Topics discussed include interrupt handling for pulse width modulation (PWM) generation and sensor processing with performance measurement for CPU bandwidth usage. Implementation of a speed profile (speed vs. time) and its interface with the interrupt handler are also described.

Part 2: Introduction

In Part 2 of this seminar, we build on the fundamentals of BLDC motor operation and control covered in Part 1, turning our attention from six-step 120-degree modulation to an examination of 180-degree modulation. We also discuss sinusoidal modulation and look at an example of code used to generate a sine wave. We then discuss open-loop V/f control and closed-loop control. The seminar ends with an overview of vector control.

180-degree modulation

Recall that for six-step 120-degree modulation, power switches are turned on and off so that the current passes through two coils, as shown in Figure 28. Every 60 degrees we switch connections so that the current flowing from coil U to V now flows from U to W. This switching effectively keeps the V coil free of current for the next 60 degrees. During this period when no current flows in the V coil, the back-EMF signal generated by the rotor's magnetic field can be detected in the V coil. The progression of six-step coil energization is shown in Figure 29.

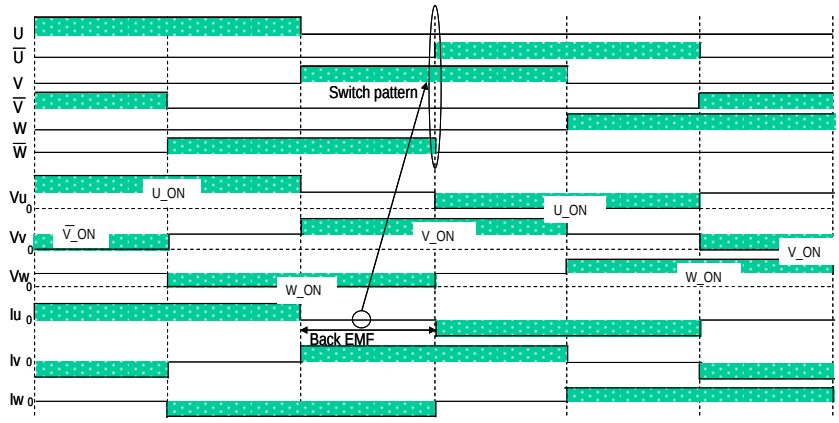


Figure 28. Six steps of trapezoidal control method.

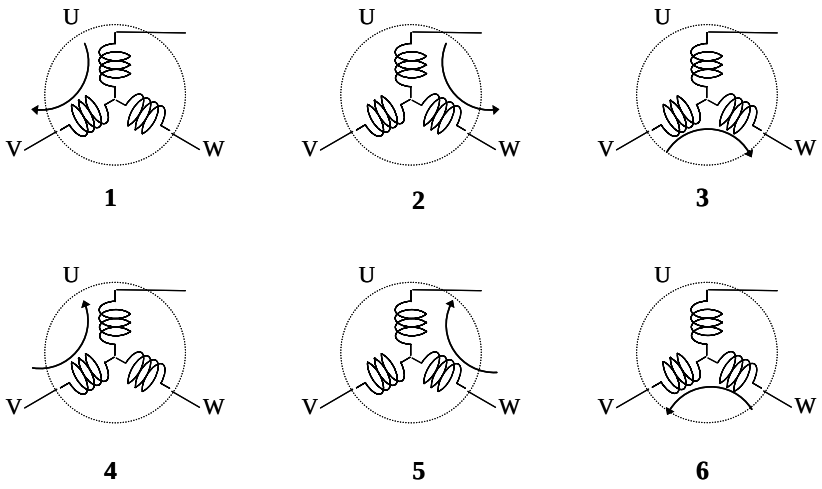


Figure 29. Progression of six coil energization steps.

Another way to understand 120-degree modulation is to look at the timing of Up, Vp, and Wp during electrical rotation. Each phase is energized for a time that corresponds to 120 degrees of electrical rotation. Phase Up is on for 120 degrees; Vp is on for the next 120 degrees; and finally Wp is on for the rest of the cycle. Lower switches are turned on and off to provide the path for current flow. In this scheme, there is a period of 120 degrees of rotation in which, for example, phase U does not create any torque on the rotor. Effectively each coil is utilized at only 2/3 of its capacity.

However, if we could use each coil for the entire electrical period, we would be able to generate more torque on the rotor. So, rather than turning on Up for 120 degrees of rotation and then waiting another 60 degrees before turning on Un, what if we could keep Up on for an entire 180 degrees of rotation with no long wait period before turning Un on? This modulation scheme is viable, but only if we provide enough transition time between the act of turning Up off and turning Un on to protect the switches from a short circuit. If we turn Un on before Up has been properly turned off, we risk creating a short circuit that can explode the power switches. The transition time required to safely implement this 180-degree modulation scheme is known as dead time.

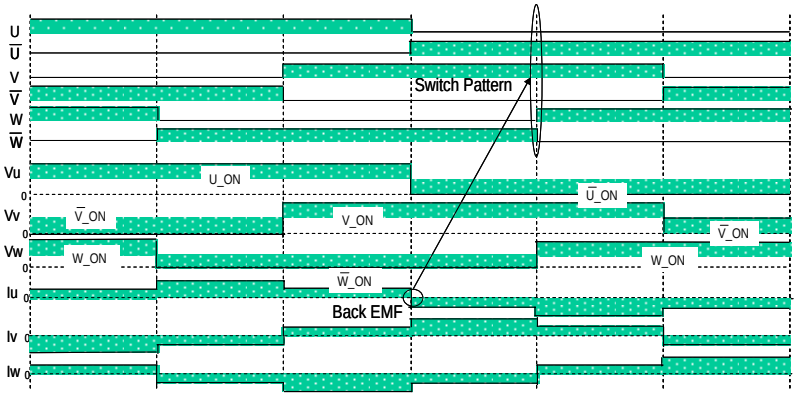


Figure 30. 180 Deg modulation scheme using three coils at a time.

Effectively with 180-degree modulation we are passing the current through all three coils at all times, inserting a small transition time to protect the power switches each time the direction of the current is switched. For the first 60 degrees, current flows in from Up and from Wp and exits the Vn coil as shown in Figure 30. The Wn coil is no longer free, as it now also passes the current. Next we switch Wp and Wn so

that the current direction changes, and for the next 60 degrees, current flows in from Up and exits the Vn and Wn coils. We continue in a similar fashion for the next four steps. In this way, the coils are utilized fully at all times to create torque on the rotor. Note that the ability to detect back-EMF is greatly diminished because of the short dead time. Generally we say that with 180-degree modulation, there is no back-EMF detection. This statement is correct for all practical purposes. The dead-time requirement also affects the MCU timers, as a dead-time register must be used to insert the proper delay before each phase can be turned on.

We can contrast 120-degree modulation with 180-degree modulation as follows. The 120-degree technique uses only 2/3 of the electrical period to create torque and rotate the motor, whereas the 180-degree modulation scheme uses the entire electrical period. Torque created using 120-degree modulation contains ripples, because torque is applied to a coil for the first 120 degrees, is not applied for the next 60 degrees, and then is applied again for 120 degrees. However, because 180-degree modulation does away with the long (60-degree) wait period, the current flow is smooth and torque ripples are mostly eliminated. Moreover, 180-degree modulation makes possible various other modulation strategies such as sine modulation, quasi-sine modulation, and space vector modulation.

Let's consider the example of a timer with a dead-time register. The Renesas M16C/Tiny series microcontroller unit (MCU) has a special 3-phase timer, shown in Figure 31, which inserts the dead time required between turning the Up and Un power switches on and off. Timer channel B2 generates the carrier frequency and Timer channels A1, A2, and A4 are used with buffers to set the pulse width modulation (PWM) values for on and off counts. The first buffer holds the value that turns on the output when a compare match occurs. The second buffer holds the value to turn off the output on a compare match. When the dead-time register is programmed with a value, two internal signals—P for Up and N for Un—are modified by inserting the dead-time count in the compare match. Then, negative signal N is turned off for Un, dead time is inserted, and positive signal P is turned on for Up. Thus our internal timer hardware makes sure that the upper and lower switches are protected properly.

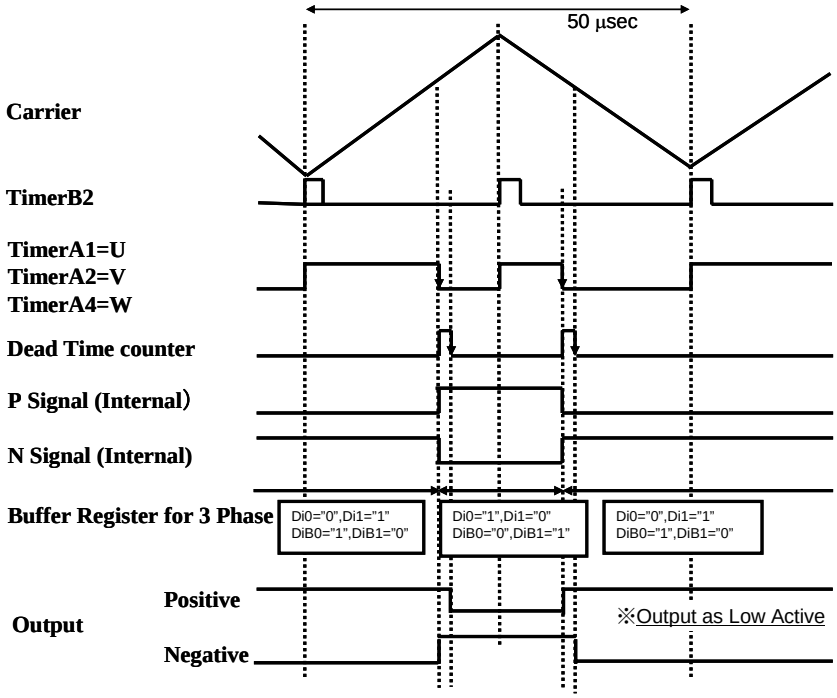


Figure 31. M16C 3-phase timer automatically inserts required dead-time.

The dead-time register is programmed once at the beginning of operation, and the count value is dependent on the characteristics of the power switches. Two internal buffer bits—Di0 and Di1—are programmed to

generate a specific high or low output on the compare match. The compare-match output is high if the bits Di0 and Di1 are set to 0,1. The output is low if these bits are set to 1,0. For the second buffer, the bits are DiB0 and DiB1. By changing the bits properly, a center-aligned or edge-aligned PWM output can be generated easily. Since each channel has a set of buffer bit settings, designers can change the behavior of any channel at will.

The MCU's 3-phase timer is quite versatile in its ability to generate various modulation schemes. It can generate 180-degree, sine wave, quasi-sine wave, space vector, or any custom modulation scheme. It can also generate 120-degree modulation with 60-degree or 120-degree modulation time. It can modulate the upper switches only, the lower switches only, both switches together, or one at a time every 60 degrees. Because the phase timer has dual buffers—the first buffer for the rising-edge compare match and the second buffer for the falling-edge compare match—dual sampling of the angle is possible for better sine-wave generation.

Sine-wave generation

Using the MCU's 180-degree modulation capability, we can generate a sine-wave output easily. In this case, instead of a 360-degree electrical cycle period, we employ a period called the carrier-wave-frequency period. During this period, we calculate sine of the angle and set the PWM accordingly. The basic steps to generate this sine-wave output, illustrated in Figure 32, are as follows:

1. Select a carrier wave frequency **fc** such as 20kHz or 16kHz.
2. Select the voltage **V0** and frequency **f** of the output wave.
3. Compute the phase angle **Θ** of the voltage at every carrier-wave period.
4. Look up the corresponding sine value from the table.
5. Multiply the sine value with the modulation ratio to generate the PWM value.
6. Transfer the PWM values to the registers.

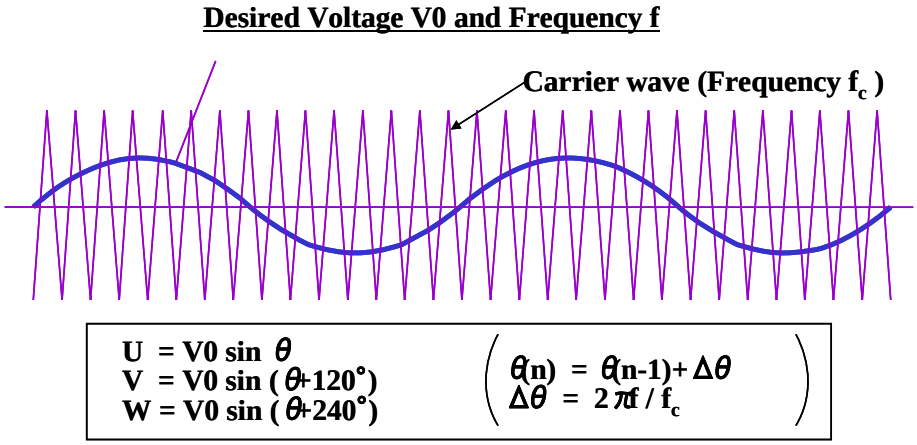


Figure 32. Basic steps of sinewave generation.

The basic formulas are listed below.

$\Delta\Theta = 2\Pi f / fc$ (computed once for frequency f)
$\Theta(n) = \Theta(n-1) + \Delta\Theta$
$U = V0 \sin(\Theta(n))$
$V = V0 \sin(\Theta(n+240))$
$W = V0 \sin(\Theta(n+120))$

Let's examine in detail the steps for sine-wave generation. Three values are required—the carrier frequency **fc**, the sine wave frequency **f**, and the voltage level **V0**. We will use fc =10kHz, f =50 Hz, and V0=100 %

of the possible DC bus voltage. The maximum voltage level is Vdc and the minimum voltage is zero, thus implying

Vmax = Vdc and Vmin = 0.

The sine wave is generated from a center value to a maximum value **V0** and then to a minimum value **-V0**. Therefore the center value is Vdc/2 computed as

(Vmax + Vmin)/2 = ½ Vdc.

Note that V0 (at maximum 100%) is also ½ Vdc. In our case, Vdc = 160 volts; therefore, V0 = 80 volts and the center point is also 80 volts. The sine wave is now calculated as

½ Vdc + ½ Vdc * Sin (Θ).

We can write this as,

Vpwm = ½ Vdc + ½ Vdc * Sin (Θ).

We will get Vpwm = Vmax at 90 degrees and Vpwm = Vmin at 270 degrees as expected.

The angle traversed every carrier frequency is now

$\Delta\Theta = 2\Pi f / fc = 360 * 50 / 10000 = 360 / 200 = 1.8$ degrees.

Also,

$\Delta t = 1/fc = 1/10000 = 100\mu s$ period,

Max PWM count = Δt * counting frequency, and

PWM max = Δt * 20MHz = 100 * 20 = 2000.

Because we are using a center-edged PWM generation timer, the timer's B2 channel is ½ of this value. The timer B2 period is 1000 counts, and the first PWM for the rising edge is

PWM1 = 500 + 500 * Sin (Θ).

We can compare this to

½ Vdc + ½ Vdc * Sin (Θ).

The second PWM for the falling edge is

PWM2 = 1000 – PWM1.

At 90 degrees, the sine of the angle is 1. We must therefore get the maximum number of counts. At 270 degrees, the sine of the angle is -1 and we must get the minimum number of counts.

In summary, to generate a sine wave, we begin with Θ=0 and then set the timer channel B2 to a value of 1000 counts for the count down. We set the interrupt at every second underflow, which is the completion of the triangular center-edge waveform. In the interrupt routine, we compute the angle, look up the sine table, and multiply by the voltage value to compute the PWM1 value. We check this value for maximum and minimum bounds and then compute PWM2. We will examine the code for this procedure in the next section.

You probably noticed that we performed a sine-table lookup in generating PWM. How do we create this sine table? Since the MCU performs all calculations in integer arithmetic, we must use a scaled value for the sine table. We also must use a scaled value for the voltage. Using Microsoft Excel, we can create a sine table with one-degree resolution similar to the one in Figure 33. We start at zero degrees and advance the table by one degree for each entry, using the sine of the average angle for the index. All sine values are computed in floating values, as shown in the third column. We then convert these values to 2^13 format—that is, value 1 is represented by 2^13 = 8192. Next we take the floor value of the calculation to get the sine value in integer format. For the U value, we continue with the angle computation, and for the V and W values, we add a 240-degree and 120-degree offset, respectively, for the lookup process. Then we plot the three values to see if correct sine waves result.

Index	Mid point angle	Sine value	Sine in 2^13 format	Integer value for Sin
1	0.5	0.008726535	71.4877788	71
2	1.5	0.026176948	214.4415605	214
3	2.5	0.043619387	357.3300213	357
4	3.5	0.06104854	500.1096359	500
5	4.5	0.078459096	642.7369122	643
6	5.5	0.095845753	785.1684046	785
7	6.5	0.113203214	927.3607272	927
8	7.5	0.130526192	1069.270567	1069
9	8.5	0.147809411	1210.854696	1211
10	9.5	0.165047606	1352.069987	1352
11	10.5	0.182235525	1492.873425	1493
12	11.5	0.199367934	1633.222119	1633
13	12.5	0.216439614	1773.073317	1773
14	13.5	0.233445364	1912.384421	1912
15	14.5	0.250380004	2051.112993	2051
16	15.5	0.267238376	2189.216777	2189

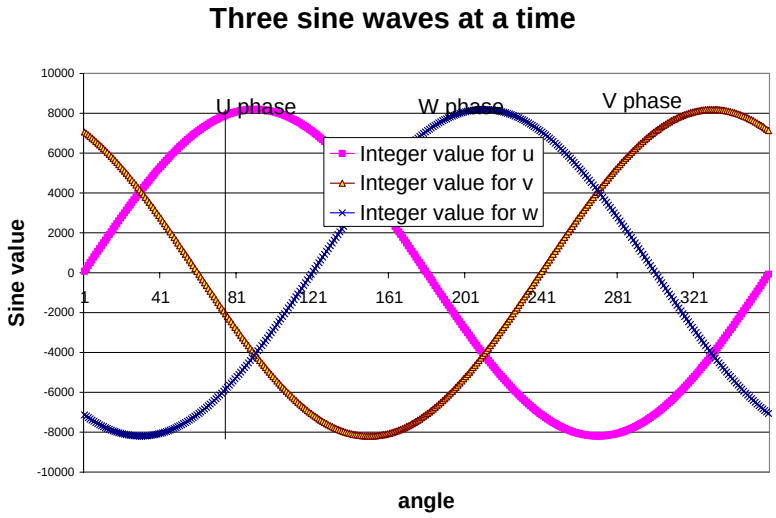


Figure 33. Sine wave table and its graph with integer values.

Several important points should be noted. Applying a dynamically changing PWM voltage to the stator results in sine-wave voltage and current passing through the stator. This action requires a carrier frequency. We select the value of the carrier frequency based on how precise and accurate the applied sine wave must be and how large a table our MCU's memory can handle. Typical carrier frequency values falls in range of 2 to 20kHz. However, many designers want to avoid the audible frequency range, so they select values of or above 16kHz.

Because a single PWM signal cannot create negative current, a lower power switch is used to enable the current to flow in the opposite direction. Positive and negative PWM output switches are required. To protect these switches, dead time must be inserted between positive and negative signals, as we have discussed previously. Hardware-based dead time is more accurate than software based dead time and reduces CPU bandwidth usage.

With sine-wave implementation, we can improve control performance and efficiency. However, this process requires a true 3-phase timer unit with dead time insertion for proper operation. Sine-wave modulation is compared to trapezoidal modulation in Figure 34.

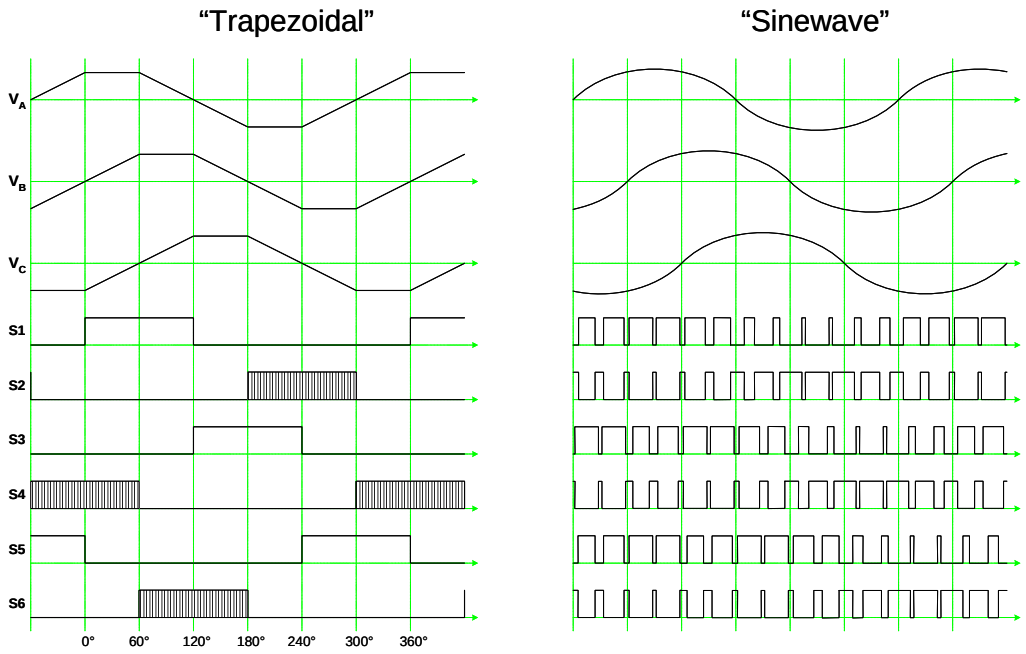


Figure 34. Comparison of 120 deg trapezoidal and 180 deg Sine wave modulation.

Sine-wave generation example

Now let's consider how to code a device for sine-wave generation. We begin with a motor-control reference platform, shown in Figure 35. This platform consists of two boards—a Starter Kit Plus (SKP) and a Power Board. A Renesas M16C/28 series MCU, an LCD, and LEDs are mounted on the SKP. The power board has AC-to-DC conversion and an integrated power module (IPM) with six power switches plus drivers built in. The IPM uses a heat sink, as Figure 35 shows.

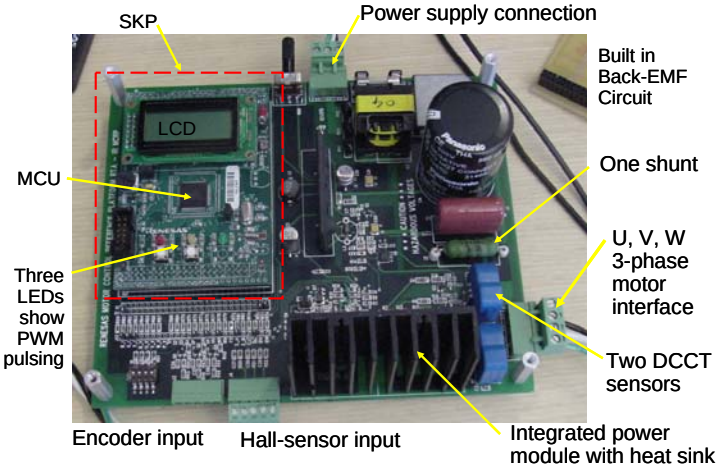


Figure 35. Power and MCU board for controlling the motor.

To measure the motor position and current in the U and V phases, the power board provides a Hall-sensor input, an encoder input, and two DCCT devices. Three back-EMF resistor ladders plus a fourth resistor ladder for measuring the Vbus are also implemented on the board, too. A precision shunt resistor on the low voltage side is also included to measure the overall current or to perform a one-shunt current detection technique for current measurements. Together these elements allow us to run various BLDC algorithms.

Our test set-up consists of a Bodine BLDC motor with two pole pairs, as shown in Figure 36. The MCU is used to generate a sine-wave PWM series of various frequencies, and the current waveforms are captured by the DCCT sensors. (You can view the entire code for the interrupt in Appendix A.) The basic sine wave is generated as follows.

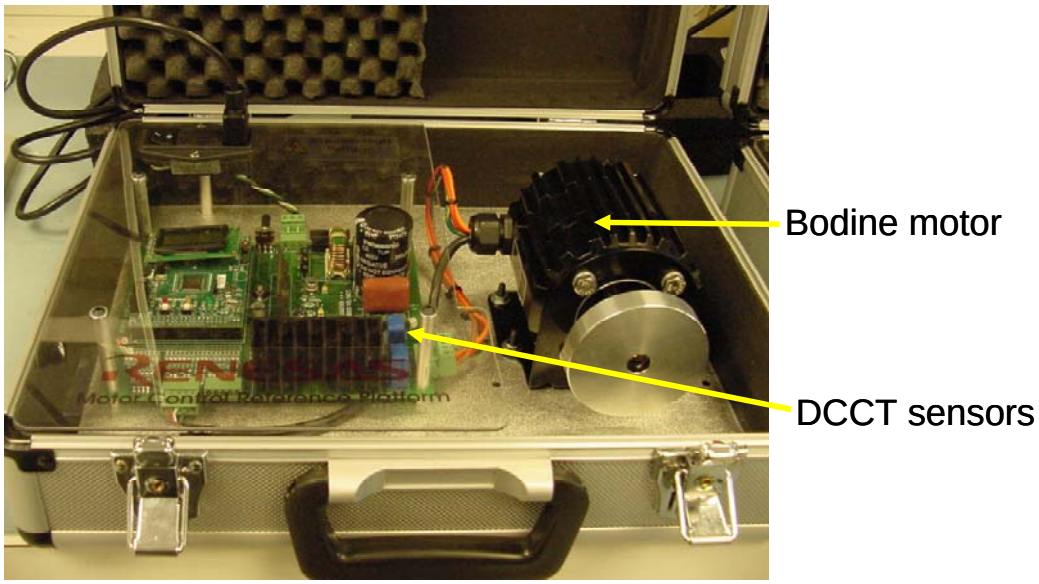


Figure 36. Test set-up with Bodine motor.

When the interrupt is entered, the MCU's firmware checks the validity of the new frequency update. If the update is true, the new frequency is commanded and the delta theta for angle computation is updated. Delta theta is presented in 2^6 format, so **delta theta = 64** means 1 degree, and **delta theta = 96** means 1.5 degree. The firmware then integrates the total angle, denoted as **sinpt_sum**, by adding the delta theta value. If the resulting sinpt_sum is greater than 360 degrees (23040 using 2^6 format), then roll-over has occurred and the sinpt_sum must be corrected to a new value by subtracting 23040. Then, sinpt_sum is scaled back by a 2^6 value to get an index in the sine table. This index is called **sin_pt** in our code.

The constants C4_DAT and C2_DAT in our code are based on carrier frequency and represent ¼ and ½ counts for the carrier-frequency time period. For example, assume that the carrier frequency is 10kHz. Thus, the time period is 100µs. Counting at 20MHz, it represents 2000 counts. It follows that C4_DAT = 500 and C2_DAT = 1000 counts. Using the C4_DAT constant, three PWM values are computed, as shown in Listing 1.

As you can see, sin_pt is used as an index for the sine table. Voltage is represented by the variable **tqdat**, and the entire multiplication is scaled by 2^19 because sine values are given in 2^13 format and voltage values in 2^6 format. For V and W values, **offset_v** and **offset_w** variables are used. To rotate the motor forward, V and W must have offsets of 240 and 120, respectively. To rotate the motor in reverse, V and W must have offsets of 120 and 240. Notice that this procedure is not unlike running the six-step state table in reverse order.

Listing 1

```
/*U-phase pwm command value = carrier/4 - (sinN° × (torque command value × carrier/4))*/
pwm_u_w = C4_DAT - (signed int)(((signed long)sin_tbl[sin_pt]*(signed long)tq_dat)>>19);

/*V-phase pwm command value = carrier/4 - (sinN° × (torque command value × carrier/4))*/
pwm_v_w = C4_DAT - (signed int)(((signed long)sin_tbl[sin_pt+offset_v[direction]]*(signed
long)tq_dat)>>19);

/*W-phase pwm command value = carrier/4 - (sinN° × (torque command value × carrier/4))*/
pwm_w_w = C4_DAT - (signed int)(((signed long)sin_tbl[sin_pt+offset_w[direction]]*(signed
long)tq_dat)>>19);
```

Next, PWM values for U, V and W are checked for PWM_MIN and PWM_MAX to protect the timer operations, and code is developed to reduce the execution time. A PWM2 value is computed using the constant C2_DAT, again to minimize the execution. Finally the six timer registers are loaded, with ta4 and ta41 representing the PWM1 and PWM2 for U phase. Since we have two buffers, we can integrate the angle for the first-half period and use the first index for PWM1. Then we can integrate the angle for the second-half period and use that index for PWM2. Although this method doubles the number of sine lookups and thus increases execution time, it also provides a higher-resolution sine wave.

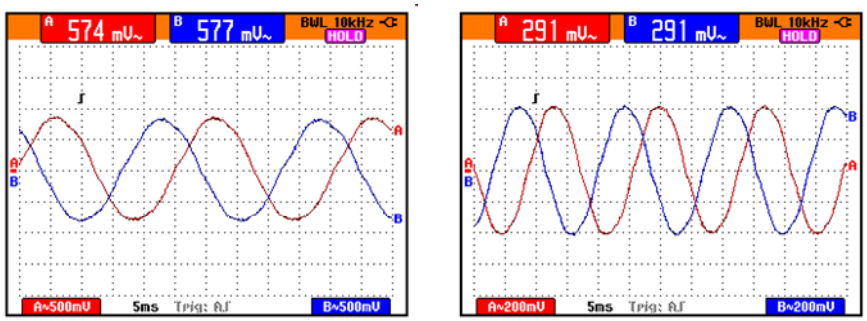


Figure 37. Current wave forms using DCCT sensors at 40 and 60 Hz sinewave.

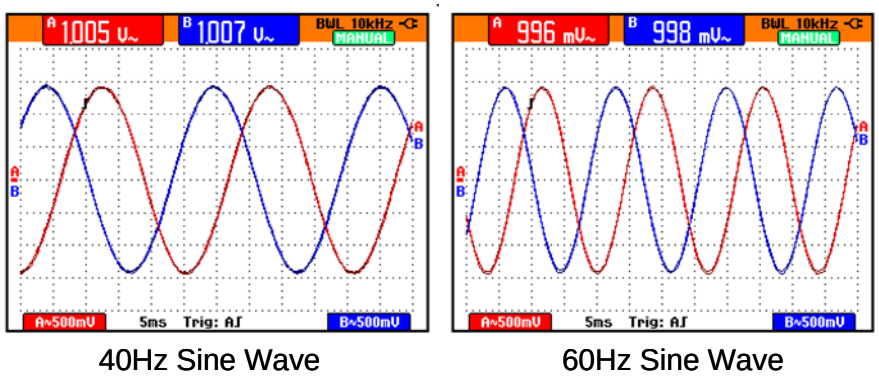


Figure 38. PWM output with 10kHz filtering.

As Figure 37 shows, we capture the current waveforms at 40Hz and 60Hz using DCCT sensors with 10kHz filtering. As Figure 38 shows, we capture the PWM output from the MCU with 10kHz filtering to view the PWM. We see that the DCCT current waveforms and PWM output are sinusoidal and represent proper fidelity.

V/f open-loop control

Voltage-over-frequency (V/f) open loop control is based on three assumptions:

1) Motor impedance increases as the frequency increases.
2) A fixed amount of current is most desirable.
3) Motor speed can be increased easily by increasing the frequency and related voltage.

Typical V/f control is run from table-based values of for voltage (the tqdat variable in our code) and frequency (delta theta variable in our code), as illustrated in Figure 39. Notice that low frequency values require lower voltage values. Generally a motor has a minimum speed called ω_{min} and an operational speed ω_{ops} at which the voltage reaches 100%. When the commanded frequency increases beyond the Wops value, the only possible control is to increase the applied synchronous frequency while holding the voltage level steady at 100 %. Table values from Wmin to Wops, as plotted on the chart in Figure 39, are determined in the laboratory by observing the current, which is generally kept constant. When the frequency increases beyond the Wops value, the current value decreases. Most of this decrease in current comes from the generation of back-EMF current by the rotating magnetic field. At Wmax speed, current becomes minimal. Generally we do not drive the motor beyond this maximum speed.

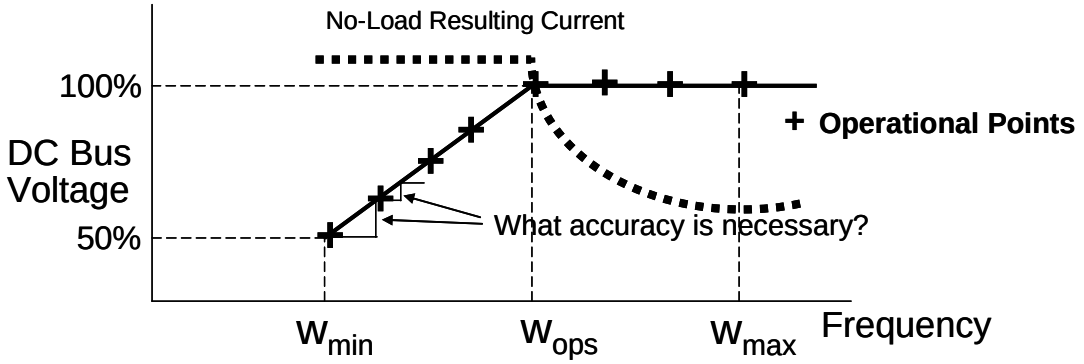


Figure 39. V/f open loop control with current behavior and operational points.

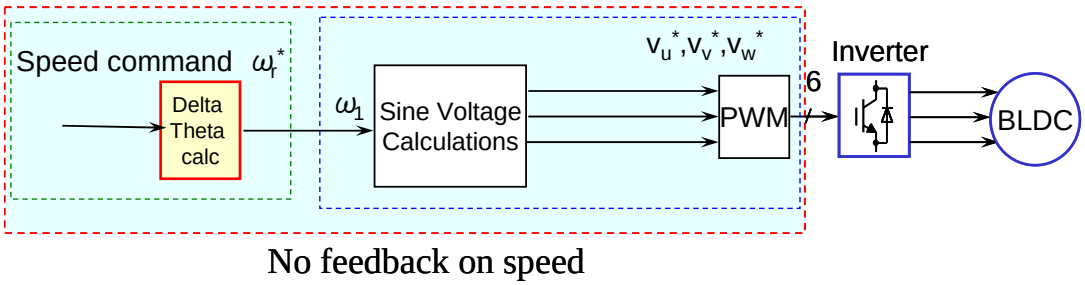


Figure 40. Firmware flow for open loop V/f control without any speed feedback.

V/f open-loop control, illustrated in Figure 40, starts with a desired motor speed. Firmware looks up the table values to set the command frequency and pre-determined voltage level. The timer is initialized and three sine waves are generated at the commanded frequency and voltage level. An interrupt routine handles the sine wave generation using PWM values. No feedback is given regarding the motor speed; we assume that the motor is running at the speed desired. This type of control is known as open-loop control—no closed loops whatsoever are used. Generally, Wmin and Wmax depend on the motor plus load and Wops is determined by the system configuration.

This control method has two main advantages. First, it requires no measurement of current or speed. Second, it uses a simple algorithm that is easy to implement. With some experimentation, we can rotate any BLDC motor without knowing the details of its parameters. This simplicity, however, also has its disadvantages. Without feedback, we do not know at what speed the motor is running. If the load is variable, the motor speed will be variable also. If the system requires some form of speed control, a speed sensor can be added. However, this moves us to a closed-loop control system.

The open-loop method provides no reading of the current, so an overcurrent condition is possible. To protect against this condition, a shunt current resistor can be used to measure the steady-state current. This technique gives some feedback regarding speed as well as overcurrent. It may also give some feedback about the load on the motor. Adding a shunt current resistor is relatively easy and again moves us to a form of closed-loop control.

Open-loop implementation example

In Figure 41 we have implemented open-loop control to show a profile with three-speed motor operation. We start the speed profile at 5 seconds by commanding a low speed of 2400 RPM. We use a ramp in speed, shown in Figure 42, to reach the commanded speed and we also use a ramp in voltage. We begin generating a sine wave with a commanded value of 1200 RPM (20Hz) and voltage (torq) value of 50% (32), because voltage is scaled 100% using a 2^6 format. During the interrupt processing, we create a “near zero” flag when the angle (or sin_pt) is near zero. In the main routine, when the near zero flag is true, firmware changes the commanded speed by a small amount—for example, 120 RPM—and also increases the torq value by 1. Based on the speed and torq start and stop values, different ramp profiles will be generated.

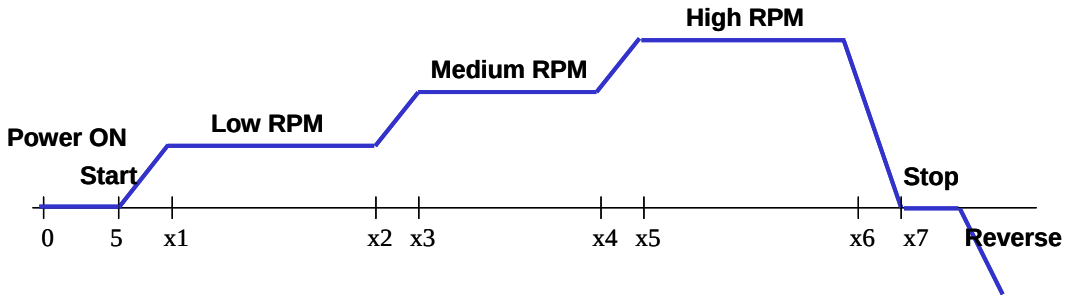


Figure 41. Speed profile with three different speed values as a function of time.

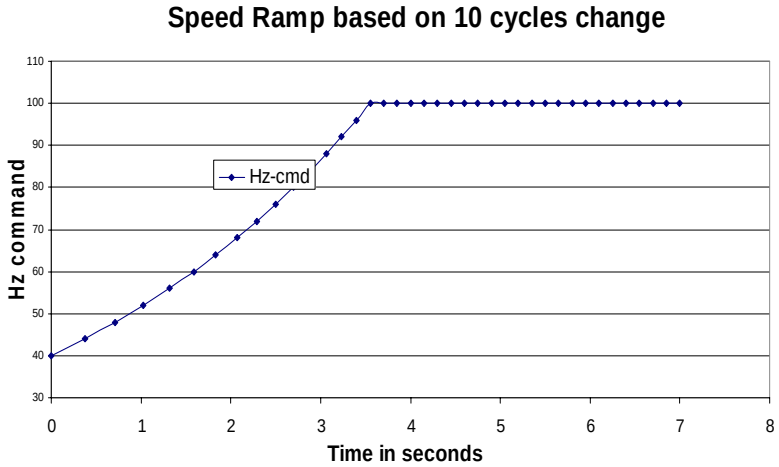
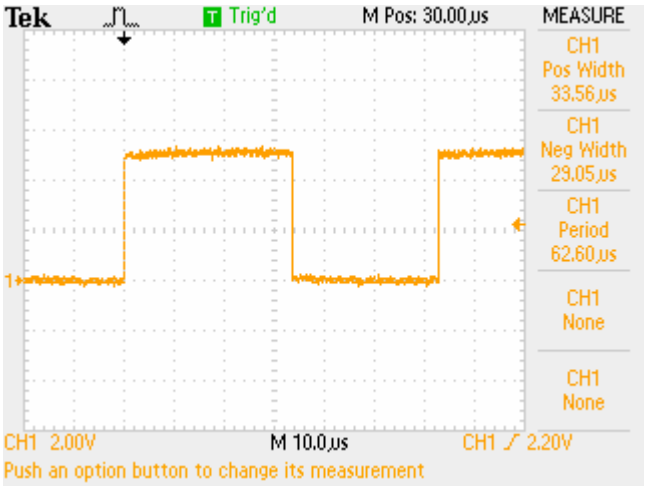


Figure 42. Speed ramp from 40 to 100 Hz as a function of time.

The motor is run at low speed for 60 seconds and then the speed is increased to a medium value of 3000 RPM. The motor reaches medium speed using the same ramp module. Then the motor is run for an additional 60 seconds and the commanded speed is increased to its high speed value of 3600 RPM. We run the motor for 60 seconds at high speed, then slow it down to 1200 RPM and stop it. We wait for 5 seconds, change the index offset, and start the motor again. This time we run the motor in reverse—an important step, because we want to be sure that the motor has this ability.



Within the interrupt code, we use a port pin to output high (1) and low(0) state. When we enter the interrupt processing module, we output high (1) state on this port pin. When the interrupt processing is complete, we output low (0) state on the pin. In this way we can measure the CPU bandwidth, as Figure 43 shows. The rising-edge to rising-edge time is our carrier-frequency time period, and the rising-edge to falling-edge time is the time it takes the firmware to execute the interrupt-processing routine.

Figure 43. CPU bandwidth measurement for open loop sinewave.

At a 16kHz carrier frequency, the measured interrupt time is 62.60µs, whereas the calculated value is 62.50µs. Our result is within the measurement error. The execution time is measured as 33.56µs, which gives us a CPU bandwidth usage over 50%. Thus more than half of the available CPU processing time is used in this interrupt. However, interrupt processing is the only time-critical task that the CPU has to do in this open loop implementation. Although not ideal, our open loop implementation is reasonable because it leaves about 50% of the CPU's time for performing other non-critical tasks. For closed loop, we will need to add other time critical tasks that will require us to reduce the CPU bandwidth for sine wave generation task.

Optimizing the sine code

Our next step is to optimize the code for sine wave PWM execution time for three reasons. 1) If we want to increase the carrier frequency from 16kHz to 20 or 25kHz, then, the interrupt time is 50 and 40 µs respectively. In this case, the CPU bandwidth usage will be 66% and 80% which is generally not acceptable. 2) We want to add time-critical speed and current measurement tasks, and 3) We want to add interrupt driven closed loop control code, which is a time-critical task also. Therefore, it is better to reduce the Sine wave PWM execution time early. To improve performance, we'd like to reduce the CPU bandwidth usage from its current level of over 50%. First we measure individual times for each PWM computation, including table lookup and long multiplication. We are looking for ways to avoid table lookups because they take a lot of time. A significant amount of time is also being spent in MAX-MIN checking. If we can avoid some of this computation, we will save more CPU bandwidth. After detailed analysis, we implement the following changes in our code:

1. Instead of computing the sine value for W index, we simply use the sum of the U and V sine values. This is true for one case only: when all three sine waves are 120 degrees apart. If the index difference between U, V, and W is not 120 degrees, then such replacement cannot be made. Since this technique is applicable to our implementation, we now save the time required for table lookup, long multiplication, and scaling.

2. We study the upper and lower bounds of the table as well as the torq voltage variable in our code and guarantee by design that the PWM values generated by our equations will not require MAX-MIN checking. Then we eliminate these checks. This process saves more CPU bandwidth but requires that we spend time reviewing the entire code for such a guarantee.

3. Finally, after making sure that we have generated a proper sine wave, we turn on the compiler optimization flag.

The optimized code is shown in Listing 2. Running this optimized code, we measure the CPU bandwidth again for comparison. As Figure 44 shows, the CPU interrupt-processing time is now 14.31µs, down from 33.56µs required for the original code. This is a reduction of more than 50% and thus CPU bandwidth usage is significantly lower. For a 16kHz carrier frequency, interrupt time is now 62.5µs, which translates into 22.9% CPU bandwidth usage versus 53.7% with the original code. These improvements give firmware designers the freedom to add time-critical sensor measurement and closed loop control tasks that are useful in motor control.

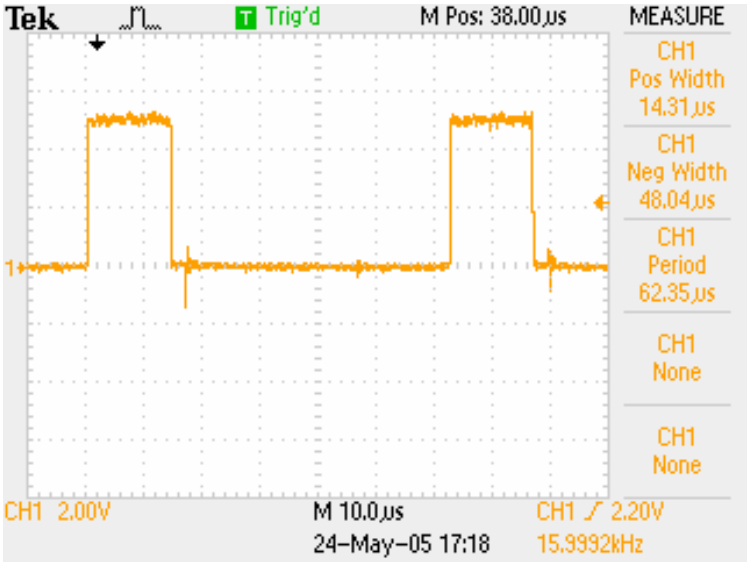


Figure 44. CPU bandwidth for optimized code.

Listing 2

```
#pragma INTERRUPT/B tb2_int
void tb2_int(void)
{
        static unsigned int dTheta;
#ifdef TIME_PWM
    p7_0 = 1;
#endif
        if(Update) {
                dTheta = DeltaTheta;
                Update = FALSE;
        }
    sinpt_sum = dTheta + sinpt_sum; /*Sine pointer sum .. sine skip-read value + sine
pointer sum */
    if(sinpt_sum > 23040) {      /*Sine pointer sum max. value? 23040 = 360° × 64 */
                NearZero = TRUE;
      sinpt_sum = sinpt_sum - 23040;    /*Sine pointer sum max value revision sin*/
        }
    sin_pt=sinpt_sum >> 6;            /* sine pointer sum / 64 */
/*U-phase pwm sine value =  (sinN° × (torque command value × carrier/4))*/
    pwm_u_w = (signed int)(((signed long)sin_tbl[sin_pt]*(signed long)tq_dat)>>19);

/*V-phase pwm sine value =  (sinN° × (torque command value × carrier/4))*/
    pwm_v_w = (signed int)(((signed long)sin_tbl[sin_pt+offset_v[direction]]*(signed
long)tq_dat)>>19);
/*W-phase pwm Sine value = (sinN° × (torque command value × carrier/4))*/
        pwm_w_w = -(pwm_u_w + pwm_v_w);// - C4_DAT;
/*  deleted the checks on MAX and MIN  ------------------------------------*/
    ta4  = (unsigned int)(C4_DAT - pwm_u_w);
    ta41 = (unsigned int)(C4_DAT + pwm_u_w));
    ta1  = (unsigned int)(C4_DAT - pwm_v_w);
    ta11 = (unsigned int)(C4_DAT + pwm_v_w));
    ta2  = (unsigned int)(C4_DAT - pwm_w_w);
    ta21 = (unsigned int)(C4_DAT + pwm_w_w));

#ifdef TIME_PWM
    p7_0 = 0;
#endif   ------------------------------------------------------------
```

Closed-loop scalar control

Since we have optimized the sine-wave implementation, let's now investigate closed-loop control using a scalar formulation. Closed-loop control requires some type of position sensor that can measure speed, as shown in Figure 45. We can get the required position feedback from a single Hall-sensor that gives one pulse per mechanical rotation, a tachometer sensor that outputs eight pulses per mechanical rotation, or from a Hall commutator that gives six signals per electrical rotation.

An input-capture function coupled with the proper counter provides two measurements: elapsed time from one input capture to a second input capture, and change in rotor position. Both measurements together provide the speed measurement. As we can infer from Figure 45, the rotor position can be detected at every input capture, and using the previous input capture data, the speed of the rotation can be computed.

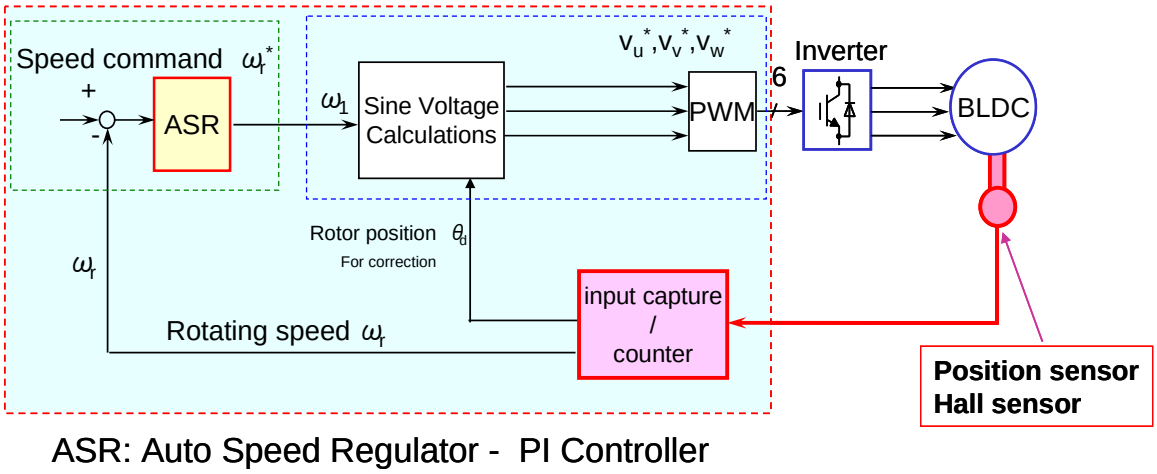


Figure 45. Closed loop speed control using position sensor and input capture timer.

Since we obtain the desired speed or speed command from a pre-set profile, it is easy to implement a proportional-integral (PI) speed regulator as shown below.

$$\Delta\omega 1 = Kp*(\omega r^* - \omega r) + Ki * \Sigma (\omega r^* - \omega r)$$

Where ωr^* = speed command,
ωr = speed measured,
Kp = proportional gain,
Ki = integral gain, and
Σ is the sum over pre-determined range.

Then, the new speed or frequency $\omega 1$ for sine-wave generation is $\omega 1 = \omega 1p + \Delta \omega 1$, where $\omega 1p$ is the previous value of the same parameter.

A similar algorithm is deployed for the new voltage value of the sine wave. Thus, two parameters are computed for each speed measurement: a new sine frequency and a new sine voltage. In many designs, voltage calculations are not implemented because the motor at that point is already running at the operational voltage and there is no need to change it. In such cases, eliminating the voltage calculations saves CPU bandwidth.

Another form of control is based on the proportional-integral-derivative (PID) algorithm. In this method, a derivative term is added to the equation using the change in the error term.

$$\Delta\omega 1 = Kp*(\omega r^* - \omega r) + Ki * \Sigma (\omega r^* - \omega r) + Kd * \{ (\omega r^* - \omega r) - (\omega r^* - \omega r)p \}$$

Where $(\omega r^* - \omega r)$ = error in speed,
$(\omega r^* - \omega r)$ = previous value of error, and
Kd = derivative gain.

Other terms remain the same. This method offers excellent control of acceleration and braking and is preferred by many experts. Disadvantages of PID, however, are convergence time and stability of the control. Depending on the gain values, the PID algorithm may react to small changes and continually perturb the system performance. Based on his experience, the author prefers to implement PI type control.

Hall processing

Let's take a look at the sensor processing steps and CPU time necessary to implement closed-loop scalar control. We use a single-pulse-per-rotation Hall-sensor for this implementation. A free-running, down-

counting timer is started during initialization. Then, at every Hall interrupt, this timer is stopped and the counts are moved into a variable called **New_Meas**. The timer is reset or reloaded and started again. A flag called **new_data** is set for the next task.

When the new_data flag is set, the **process_hall** module computes the speed. The module checks for conditions such as underflow or overflow and then calculates speed, based on one pulse per rotation. If we have implemented the type of Hall-sensor generally used for commutation signals, then speed is computed based on 60 degrees of electrical rotation. Next, the speed computation is converted into mechanical speed using proper transformation based on the number of pole pairs. The new_data flag is cleared for the measurement task and a flag is set for the speed regulator, which we call the velocity regulator. Once the new flag has been set, the velocity regulator processes the new speed measurement and creates new values for frequency and voltage using one of the formulas previously described.

Running closed-loop control of our BLDC motor using Hall sensors and other sensors gives preliminary measurement results such as those shown in Figures 46. The execution times for the two tasks are as follows:

Process_halls	16µs every Hall interrupt
Velocity_regulator	10µs every speed measurement ➔ every Hall interrupt

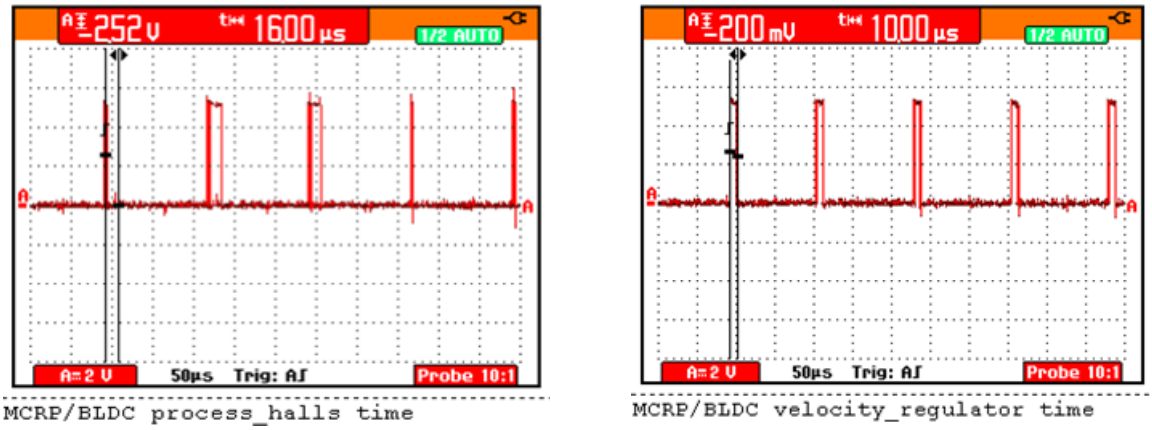


Figure 46. CPU bandwidth measurements for sensor measurement and closed loop speed control.

Now we'll analyze CPU bandwidth usage for this closed-loop control. Sine wave PWM interrupt processing time is 15µs. At a 16kHz carrier frequency, this time is 16000*15, which gives an interrupt-processing time of 240000µs. This time period, which is required to generate a proper sine wave at a given frequency, represents nearly 24% of the CPU bandwidth at the 16kHz carrier frequency. Note that the sine wave PWM interrupt-processing execution time is not dependent on the speed. This is good news, because firmware designers will not have to calculate CPU loading every time the speed changes. The CPU usage turns out to be nearly constant at every speed.

Our execution time for speed measurement or Hall process is 16µs. Let's use the case in which one Hall signal is sensed and processed per rotation. Because the number of times Hall processing must be performed depends on the speed of the motor, the processor bandwidth required for speed measurement will also be speed dependent. We will create a table, called Table IV, for two different speeds. Note that velocity- or speed-regulator execution time is 10µs, and is also speed dependent. This measurement also has an entry in our table. We compute the CPU bandwidth for two speed values—100Hz and 200Hz (indicating that each task must be processed 100 and 200 times). As shown in the last column of Table IV, the CPU bandwidth increases very slightly, from 24% to 24.2% and 24.5%. That is not much of an increase at all.

Table IV. CPU Bandwidth calculations for one sensor measurement and one control processing per mechanical rotation

Speed (RPM)	Speed (Hz)	Hall Process time in μs	Velocity regulator time in μs	Total time
6000	100	1600	1000	242600μs or 0.24 seconds
12000	200	3200	2000	245200μs or 0.245 seconds

We also want to calculate the CPU bandwidth for a case in which typical Hall commutation signals are used for speed measurements. Our motor has four pole pairs, and thus four electrical rotations per one mechanical rotation. Since there are six commutation signals per electrical rotation, the number of Hall processes has now increased by a factor of 24. We average the speed measurements for one electrical rotation and apply the velocity regulator once per electrical rotation or four times per mechanical rotation. The corresponding CPU bandwidth calculations are summarized in Table V.

In this case, sine-wave generation requires the same bandwidth as before, while speed measurement and speed control is done more often and therefore require more bandwidth. However, the increase in CPU bandwidth is still not much: bandwidth usage increases from 24% to 28% and 32 %, which is an increase of only about 4% for every 6000 RPM increase.

It's important to point out that we made our calculations to show how the CPU bandwidth usage increases. The designer must still decide how to set the speed measurements and control-loop timings. If other tasks will be impacted by the 4% bandwidth increase, perhaps they can be scaled back. This kind of tweaking may have an impact on speed accuracy, but that is what optimization is all about. As we saw previously in Table IV, we can always perform speed measurement and speed control tasks only once per mechanical rotation to reduce the CPU bandwidth.

Table V. CPU Bandwidth calculations for 4-pole-pair motor.
(24 times sensor measurement and 4 times control processing per rotation.)

Speed (RPM)	Speed (Hz)	Hall Process time 24 times per mechanical rotation	Velocity regulator time in μs 4 times per mechanical rotation	Total time
6000	100	38400	4000	282400 μs or 0.28 seconds
12000	200	76800	8000	324800 μs or 0.32 seconds

V/f open-loop control versus closed-loop scalar control

We have discussed in detail the methods for V/f open-loop and closed-loop-scalar control and evaluated the CPU bandwidth requirements for each. Now let's compare features and benefits of each type of control method. As the name implies, V/f open-loop provides no feedback regarding speed, while closed-loop control performs speed measurement using some sort of position sensor—for example, commutation Hall-sensors; a single Hall-sensor; an encoder with A, B quadrature and Z zero synch pulses; or a tachometer with multiple pulses per rotation.

In both the open- and closed-loop methods, the MCU generates sinusoidal modulation to the inverter drivers. However, speed-control accuracy differs significantly. Because the open-loop method does not provide either feedback correction or control of the speed, the accuracy is poor. With closed-loop scalar control, on the other hand, the feedback on actual speed and the corrections made to the frequency and voltage generation result in highly accurate speed control. The motor rotates very close to the commanded

speed in scalar control. No such expectation is possible in open-loop control, though, which cannot even tell us at what speed the motor is running.

Torque control is not applied in open-loop control, and it exists only indirectly in scalar control.

In terms of MCU resources, both control methods require a three-phase timer unit with dead-time insertion capability. Open-loop control requires no further resources except monitoring for high current and high temperature conditions that would call for emergency shutdown. In scalar control, however, the MCU must have an additional timer to measure the time between two position pulses, as well as input capture with interrupts. The V/f open-loop control method is relatively easy to implement at minimal cost, while the need for sensors makes scalar control somewhat more expensive.

Vector control

Another method worth discussing is vector control. The detailed formulation of this method to control the torque and flux in PMSM was originally achieved by Jahns, Kliman, and Neumann [1] in the mid-1980s. Additional work has been done by many authors, and you will find more information and explanation in references [2,3,4]. Here we will simply summarize the concept and compare it to the 180-degree V/f open-loop and closed-loop scalar control methods previously discussed.

When we examine the torque equation for a brushless DC motor, we realize that the equation is really a vector formulation with the vector product of the current and magnetic fields shown on the right-hand side:

$\boldsymbol{T \rightarrow I \times B}$, where current and magnetic field are vector quantities.

If we formulate a rotor frame that has a d-axis parallel to the north-south line and a q-axis perpendicular to the d-axis, as illustrated in Figure 47, we can actually convert the stator currents in the rotor frame. We then realize that the current along the d-axis creates pure flux, whereas the current along the q-axis creates pure torque. Denoting these two currents as Id and Iq, we can say that our control algorithm must control both these currents to maintain proper flux and proper torque in the system. We know that speed is directly related to torque, and torque is related to the q-axis current. Therefore, we must create a reference q-axis current to maintain the speed. Then we can control the q-axis current through transformation of our axes. In this way we convert the single-variable speed control into control with more variables—speed, d-axis current, and q-axis current—and we use a vector formulation to compute the quantities that we need to control. Hence we call this method “vector control.”

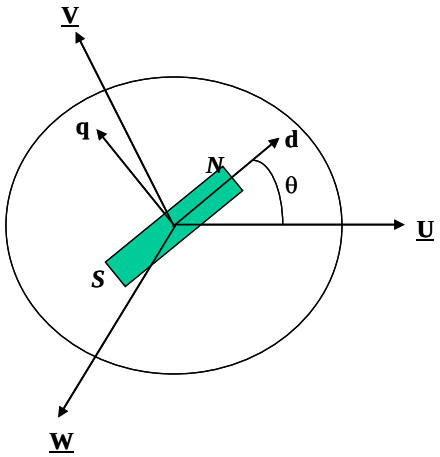


Figure 47. Stator frame to rotor frame transformation for vector control.

Now let's take a look at the detailed steps necessary for vector control. As Figure 48 shows, a profile module gives the speed command to the control algorithm at some point in time. On the right hand side, a control function outputs commands for the inverter, which is connected to the motor. This configuration has two current sensors to measure the phase U and phase W currents. The sensors are connected to the ADC on the MCU. The motor also has an encoder mounted on its rotor to give the quadrature pulses **A**, **B** and also the zero synch pulse **Z**. All three signals are sent to the input-capture and timer/counter peripheral for speed measurement.

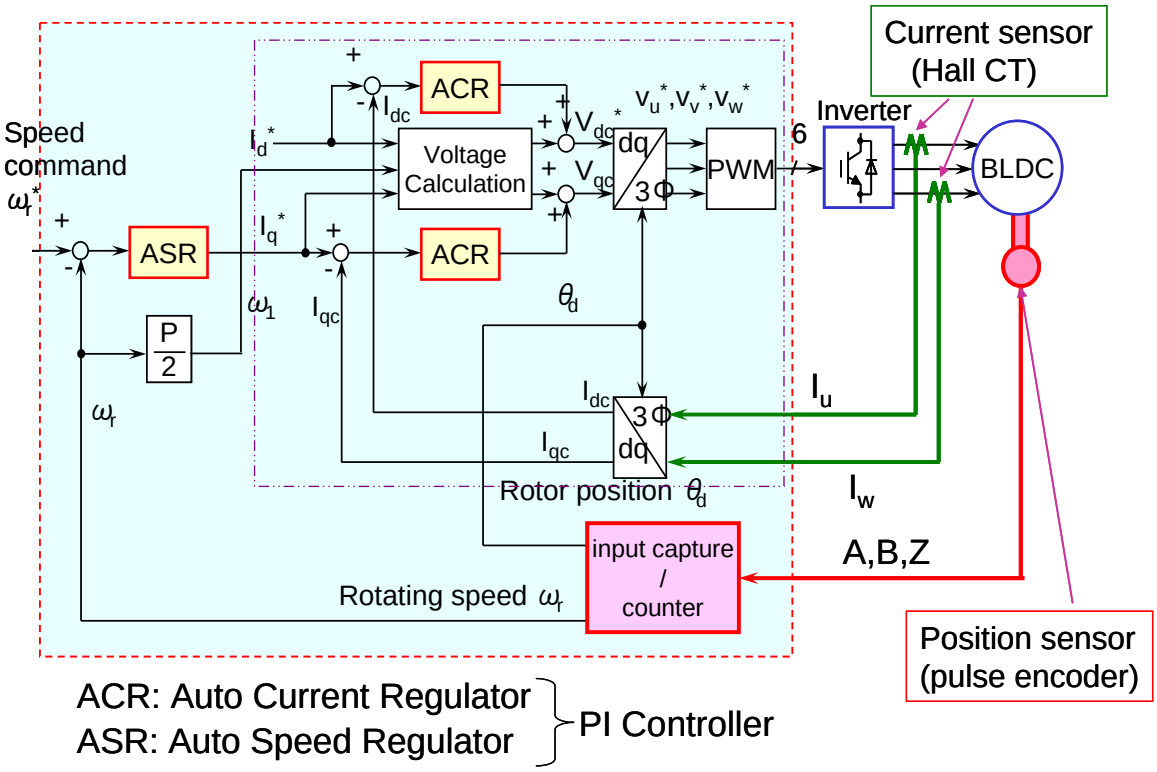


Figure 48. Vector control flow diagram.

At every carrier-frequency interrupt, three PWM signals are generated. While all three PWM signals are applied, the system measures two currents by triggering the ADC channels. During the next interrupt execution, these currents are then transformed from stator U, V, and W axes to d-q axes using matrix multiplications that involve the rotor angle q at that time. The rotor angle is measured by reading the A,B pulses and converting this reading into a proper angle. The control system's firmware is greatly helped if the MCU has a hardware timer with input capture and continuous counting of A,B pulses.

Based on the rotor position θ, stator currents are transformed in d-q axes currents as we have noted. The speed measurement is fed into the auto speed regulator (ASR), shown in Figure 48, which generates the reference q-axis current required to maintain the commanded speed. This reference q-axis current and measured q-axis currents are fed into the auto current regulator (ACR) to create q-axis voltage to be applied to the next PWM.

The reference current in the d-axis is maintained at constant level to maintain proper flux in the stator. This reference d-axis current and the measured d-axis currents are fed into a second ACR to create the d-axis voltage. Corrections are made to the voltage calculations according to the number of pole pairs and the reference currents in the d and q axes. When the final values Vd and Vq are computed, they are transformed from the rotor frame to the stator frame using inverse transformation and that rotor angle

value. Three voltages in the stator frame—Vu, Vv and Vw—are converted into the PWM values that are to be output by the three-phase timer unit.

Current measurements and auto current regulators are executed at every carrier frequency. This process, which is known as “inner loop,” uses the fastest control algorithm. In contrast, encoder measurements, and especially speed measurements, are performed at a lower rate. Therefore, the auto speed regulator and related computations are performed using a slower process called “outer loop.” A typical carrier-frequency or inner loop rate is about 4kHz or more, and the encoder-based speed computation or outer loop rate is about 500Hz or so. Occasionally the outer loop rate can drop as low as 50Hz.

Experts agree that vector control method controls the torque and flux very well, and it maintains the desired speed accurately. Vector control requires one position sensor and two current sensors to perform the necessary tasks. It also requires an MCU with high computing power so that the inner-loop and outer-loop processes can be executed properly. Additionally, the MCU must be capable of measuring two currents simultaneously, so it must have two sample-and-hold circuits in its ADC peripheral.

The vector-control method provides dynamic torque control based on exact speed measurements and current measurements. Consider an example in which the load changes during rotation. Since speed is measured several times (typically at a 500Hz rate), any load changes that affects the speed will be detected and for the next rotation, the q-axis current will be adjusted properly to maintain the same speed. If higher current is required, it will be provided. Control and system experts generally regard vector control as the reference against which the performance of other methods are compared and evaluated.

The vector-control method does have some drawbacks. It requires sensors and thus adds cost to the final implementation. Also, it mandates an MCU with high computing power, which may add cost.

In Table V1, we see a comparison of the features, accuracy, and required MCU resources for the three control methods we have covered thus far: V/f open-loop, scalar, and vector control.

Table VI. Comparison of V/f open loop, scalar and vector control.

	Features	**V/f open loop control**	**Scalar control**	**Vector control**
1	**Control method**	Open loop for speed	Closed loop for speed	Closed loop for speed and current
2	**Speed control accuracy**	Poor	High accuracy	Very high accuracy
3	**Torque control**	None	Indirect only	Optimum method
4	**MCU Resources**	3-ph timer with dead time insertion	3-ph timer with dead time insertion	3-ph timer with dead time insertion
			Input capture w/timer to measure speed	Input capture w/timer to measure speed
				2 simultaneous channels of ADC to measure phase currents
		CPU bandwidth very reasonable	Very small increase in CPU bandwidth	Very large increase in CPU bandwidth
5	**Sensors**	None	Position sensor – Hall sensor, encoder, tachometer	One position sensor and two current sensors
6	**Other notes**	Easy to implement	Speed detection is necessary	Speed and current detection necessary
			Cost for position sensor	Cost for position and current sensors
				Cost for MCU computing power
7	**Overall Score**	OK	Better	Best control so far

Sensorless control

Since the vector control method we have just examined requires one position sensor and two current sensors, the final system configuration may be costly. Consumer applications, especially white goods, are very cost-sensitive and thus cannot afford this type of implementation. At the same time, these applications do not require the same performance accuracy for speed, as do industrial applications. Therefore, two other control techniques have been developed that provide adequate performance for the system, yet keep cost down. These techniques are called sensorless because they do not require any position sensors. Both methods use the same 180-degree modulation and vector control algorithm.

The first of these methods eliminates the position sensor but keeps the two current sensors. It is known as “DCCT-based sensorless vector control” and is shown in Figure 49. Because this method uses no position

sensor, angle and speed are estimated using the current measurements and voltages applied the previous PWM cycle.

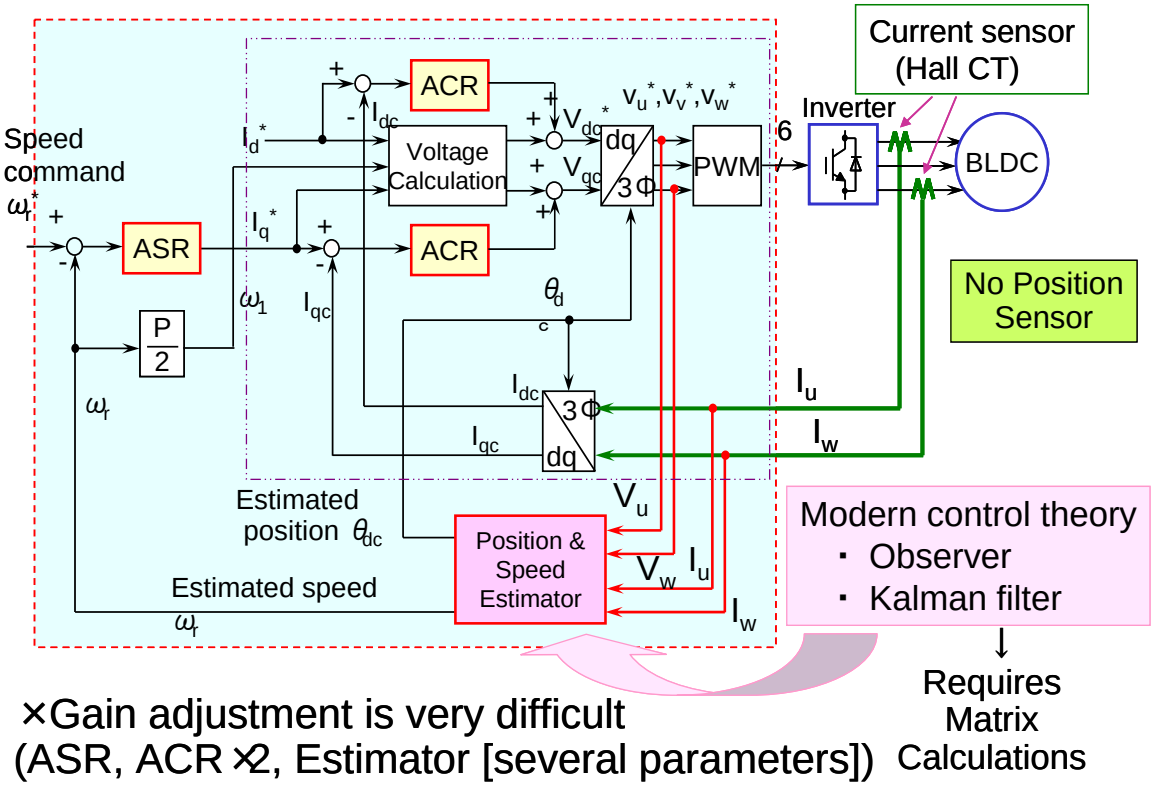
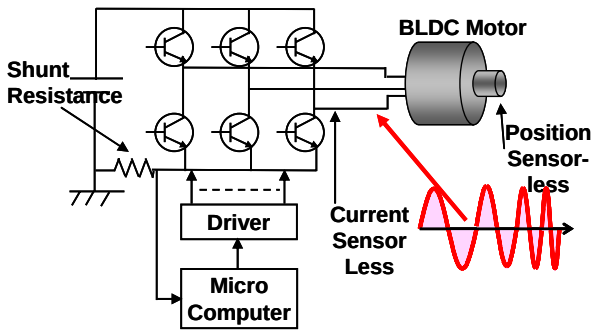


Figure 49. Position sensorless control with 2 DCCT current sensors.

The method employs a Kalman-filter approach based on principles of modern control theory, an observer-based model, and a state transition matrix. Estimated angle and speed are used together in the same vector control algorithm to control the current in the q-axis. Such an implementation requires many matrix calculations, and thus an MCU with high computing capability is a requirement. In fact, the CPU bandwidth needed is nearly double that of vector control method. Gain adjustment in the auto speed regulator and auto current regulator is very difficult. Exact motor parameters must be known, particularly the q-axis and d-axis inductance parameters, which are difficult to measure. Despite the challenges, such control is a reality and has been implemented in several applications.

6) 180-deg OSCD Sensor-less Vector Control



In a second method, the position sensor and two DCCT sensors are eliminated. Currents are measured using the shunt resistance installed on the low side of the inverter, as shown in Figure 50. This shunt resistance is a precision resistor capable of measuring the full current range of the motor. Using one shunt, we measure two currents; thus this method is known as “one shunt current detection vector control” or simply “OSCD vector control.”

Figure 50. One shunt current detection method that eliminates DCCT and position sensor.

The implementation shown in Figure 51 is similar to that of the DCCT method, but it adds one more computing block, **Current Meas**. To understand why this addition is necessary, we'll look at how the current is measured.

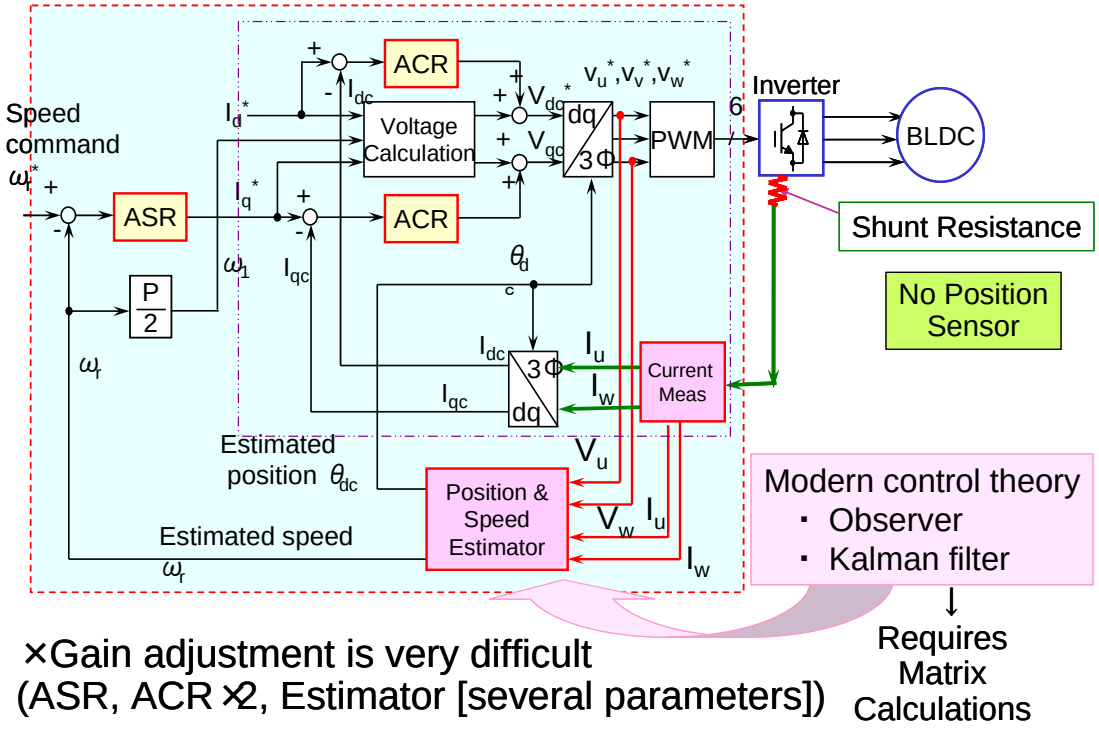


Figure 51. OSCD implementation flow with current measurement module.

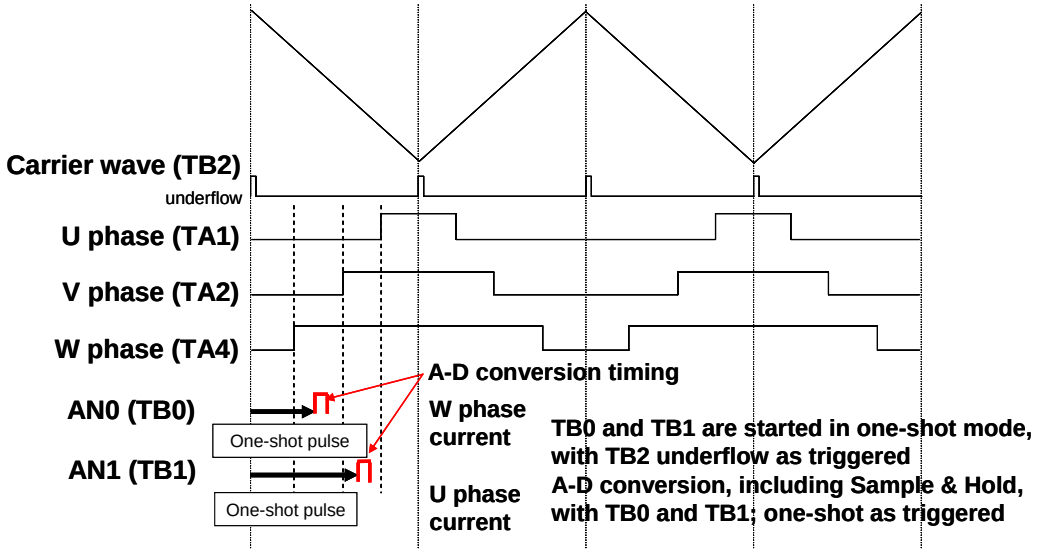


Figure 52. One shunt current measurement technique using a additional timers to trigger ADC.

First, remember that our setup has only one shunt resistor. To measure individual phase currents, we must be careful. Figure 52 shows us how the three PWM outputs are applied. In this figure, the W phase has largest PWM time, V has next smaller, and U has the smallest. If we measure the shunt current between the rising edge of W and the rising edge of V, we know that only the W phase (that is, only the Wp - upper W

phase IGBT) is on at that time. So, we measure W phase current. Next, if we measure the current between the rising edge of V and the rising edge of U, then we are measuring the W and V phase current together. This also means that we are measuring the U phase current, because the sum of all currents in a star-winding motor is zero. Thus, we must make 2 current measurements at precise time during our interrupt processing. We need two other timer channels that can help us trigger the ADC at a precise time. An example is the Renesas M16C 3-phase timer unit. This unit has a link with timer channels TB0 and TB1, such that it will trigger the ADC channels AN0 and AN1 at a precise time. All we have to do is load the appropriate register values.

As part of this process, we must compare the PWM values of all three phases and determine exactly how much time we need to load channels TB0 and TB1. It is important to note that the three PWM values continue to change constantly, so that W is not the largest. Thus, we need to test the largest value every time and set the proper flags for the current we are measuring. All the comparisons, settings, and identifications required by this method make for complex processing task, one that requires significantly more code and more CPU time. The requirements for CPU bandwidth are generally more than double those of other vector-control schemes. The more complex software and higher computing power requirements keep many designers from using this method.

Performance by both sensorless methods is adequate and useful for applications that don't have tight accuracy requirements for speed. Cost is less than full vector methods and response is good—definitely better than that provided by scalar control.

All six control algorithms we have examined, from 120-degree modulation through OSCD control, have been implemented in the Renesas M16C/28 series MCU with a prototype motor control platform. Measured performance, CPU bandwidth and code size are shown in Figure 53. As we see, vector control with position and current sensors requires about 40% of the CPU bandwidth, while DCCT sensorless control requires about 74% or nearly double the bandwidth. Moreover, OSCD vector control without position and current sensors requires nearly 90% of CPU bandwidth, which is more than double the bandwidth used by vector control with sensors.

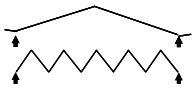
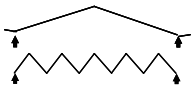
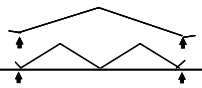
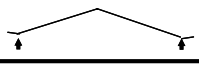
Algorithm	Carrier Freq.	ROM/RAM	Sampling Freq. (Calculating Freq.)	CPU Load Ave (Max)
120-deg Trapezoidal Wave	4KHz 20KHz	1.97KB/41B		8% (11%)
120-deg Trapezoidal Wave Sensor-less	20KHz	2.17KB/51B		30% (50%)
180-deg Sinusoidal Drive V/f	4KHz 20KHz	3.14KB/51B		12% (15%)
180-deg Vector Control – External sensor	4KHz 20KHz	4.38KB/143B		40% (42%)
180-deg (2 DCCT) Sensor-less Vector	4KHz 8KHz	6.87KB/193B		74% (75%-> 67%)
180-deg OSCD Sensor-less Vector	4KHz	10KB/1KB		88% (91%-> 91%)

Figure 53. Comparison of CPU bandwidth and code size for six speed control algorithms.

Summary

In Part 2 of this seminar, we introduced the 180-degree modulation technique and a procedure for three-phase sine-wave generation. We detailed the necessary steps for sine-wave generation, examined the code, and measured the CPU performance. We then discussed V/f open-loop control with its implementation using the M16C series device. We looked at the speed profile with three different speed settings and at a start-up ramp sequence in frequency and voltage. Next we discussed the optimization of sine-wave code and examined the CPU performance. We continued with closed-loop scalar control using a speed sensor, discussed performance of the control algorithm, and compared this method with the V/f open-loop algorithm. In the final sections, we briefly discussed vector control and the sensorless DCCT and OSCD control algorithms. We compared the CPU performance for all six algorithms, from 120-degree modulation to sensorless vector controls.

References:

1. Interior Permanent-Magnet Synchronous Motors for Adjustable Speed, Drives, by T. M. Jahns, G. B. Kliman and T. W. Neumann, IEEE transactions on Industry Applications, Vol. IA-22, No. 4, pp. 738-747, July/August 1986.

2. Dynamic Model of PM Synchronous Motors, by Dal Y. Ohm, Drivetech Inc. Blacksburg, VA

3. Modeling and Parameter Characterization of Permanent Magnet Synchronous Motors, by D. Y. Ohm, J. W. Brown and V. B. Chava, Proceedings of the 24th Annual Symposium of Incremental Motion Control Systems and Devices, San Jose, pp 81-86, June 1995.

4. Power Electronics and AC Drives, by B. K. Bose, Prentice-Hall 1986.

5. Power Electronics and Variable Frequency Drives Technology and Applications, Edited by Bimal K. Bose, IEEE Press, ISBN 0-7803-1084-5, 1997

6. Motor Control Electronics Handbook, By Richard Valentine, McGraw-Hill, ISBN 0-07-066810-8, 1998

7. FIRST Course On Power Electronics and Drives, By Ned Mohan, MNPERE, ISBN 0-9715292-2-1, 2003

8. Electric Drives, By Ned Mohan, MNPERE, ISBN 0-9715292-5-6, 2003

9. Advanced Electric Drives, Analysis, Control and Modeling using Simulink, By Ned Mohan, MNPERE, ISBN 0-9715292-0-5, 2001

10. DC Motors Speed Controls Servo Systems including Optical Encoders, The Electro-craft Engineering Handbook by Reliance Motion Control, Inc. [No ISBN number; very old book.]

11. Modern Control System Theory and Application, by Stanley M. Shinners, Addison-Wesley, ISBN 0-201-07494-X, 1978

12. The Industrial Electronics Handbook, Editor-in-Chief J. David Irwin, CRC Press and IEEE Press, ISBN 0-8493-8343-9, 1997

Appendix A Sine wave Generation Interrupt Code Example Listing.

```
#pragma INTERRUPT/B tb2_int
void tb2_int(void)
{
        static unsigned int dTheta;
#ifdef TIME_PWM
    p7_0 = 1;
#endif
        if(Update) {
                dTheta = DeltaTheta;
                Update = FALSE;
         }
    sinpt_sum = dTheta + sinpt_sum; /*Sine pointer sum .. sine skip-read value + sine
pointer sum */
    if(sinpt_sum > 23040) {       /*Sine pointer sum max. value? 23040 = 360° × 64 */
                NearZero = TRUE;
      sinpt_sum = sinpt_sum - 23040; /*Sine pointer sum max value revisionsin*/
          }
    sin_pt=sinpt_sum >> 6;            /* sine pointer sum / 64 */
/*U-phase pwm command value = carrier/4 - (sinN° × (torque command value ×
carrier/4))*/
    pwm_u_w = C4_DAT - (signed int)(((signed long)sin_tbl[sin_pt]*(signed
long)tq_dat)>>19);
/*V-phase pwm command value = carrier/4 - (sinN° × (torque command value ×
carrier/4))*/
    pwm_v_w = C4_DAT - (signed int)(((signed
long)sin_tbl[sin_pt+offset_v[direction]]*(signed long)tq_dat)>>19);
/*W-phase pwm command value = carrier/4 - (sinN° × (torque command value ×
carrier/4))*/
    pwm_w_w = C4_DAT - (signed int)(((signed
long)sin_tbl[sin_pt+offset_w[direction]]*(signed long)tq_dat)>>19);
/*U-phase PWM revision*/
   if(PWM_MAX < pwm_u_w) {             /*Duty at MAX?*/
      work_u = PWM_MAX;                /*First half .. MAX value*/
      work_u1 = C2_DAT - PWM_MAX;      /*Last half .. carrier period/2 - MAX value*/
   }
   else {
       if(PWM_MIN > pwm_u_w) {         /*Duty at MIN?*/
         work_u = PWM_MIN;             /*First half .. MIN value */
         work_u1 = C2_DAT - PWM_MIN;   /*Last half .. carrier period/2 - MIN value*/
      }
      else {                          /*MIN < duty < MAX*/
          work_u = pwm_u_w;           /* First half .. PWM command value*/
          work_u1 = C2_DAT-pwm_u_w;   /* Last half .. carrier period/2 */
      }                               /* - U-phase PWM command value */
   }
/*V-phase PWM revision*/
   if(PWM_MAX < pwm_v_w) {             /*Duty at MAX?*/
      work_v = PWM_MAX;                /*First half .. MAX value */
      work_v1 = C2_DAT-PWM_MAX;        /*Last half .. carrier period/2 - MAX value*/
   }
   else {
       if(PWM_MIN>pwm_v_w) {          /* Duty at MIN? */
```

```
            work_v = PWM_MIN;              /* First half .. MIN value */
            work_v1 = C2_DAT-PWM_MIN;     /*Last half .. carrier period/2 - MIN  value*/
        }
else {                          /* MIN < duty < MAX */
            work_v = pwm_v_w;             /* First half .. PWM command value */
            work_v1 = C2_DAT-pwm_v_w;     /* Last half .. carrier period/2 */
        }                               /* - V-phase PWM command value */
    }
/* W-phase PWM revision */
    if(PWM_MAX < pwm_w_w) {                 /*Duty at MAX?*/
        work_w = PWM_MAX;                  /* First half .. MAX value*/
        work_w1 = C2_DAT-PWM_MAX;           /* Last half .. carrier period/2 - MAX value*/
    }
    else {
         if(PWM_MIN > pwm_w_w) {            /*Duty at MIN?*/
            work_w = PWM_MIN;               /*First half ..MIN */
            work_w1 = C2_DAT-PWM_MIN;       /*Last half .. carrier period/2 - MIN value*/
        }
        else {                          /*MIN < duty < MAX*/
             work_w = pwm_w_w;              /*First half .. PWM command value*/
             work_w1 = C2_DAT-pwm_w_w;      /*Last half .. carrier period/2 */
        }                               /* - W-phase PWM command value*/
    }
     ta4  = work_u;        /*Set U-phase PWM */
     ta41 = work_u1;
     ta1  = work_v;        /*Set V-phase PWM */
     ta11 = work_v1;
     ta2  = work_w;        /*Set W-phase PWM */
     ta21 = work_w1;

#ifdef TIME_PWM
    p7_0 = 0;
#endif

}
```