To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

APPLICATION NOTE

R8C/25 Group

Driving of a 3-phase BLDC Motor by 120-Degree Trapezoidal Wave Commutation using HALL Sensors

Table of Contents:

1.0 Abstract

This Application Note shows the implementation of 3-phase BLDC motor drive by 120-degree trapezoidal wave commutation. The method shown utilizes the HALL sensors in the motor to determine the motors rotor position, effect commutation, and provide speed measurement.

This example applies to MCUs in the R8C/24 Group.

2.0 Conditions

The explanation of this issue is applied to the following condition:
Applicable MCU: R8C/24 Group (such as R8C/25 device R5F21256)
MCU operational frequency: 20 MHz
Memory size: ROM 32 KB, RAM 1 KB

Peripherals: Timer RD for PWM motor drive, Timer RB for speed measurement

3.0 Introduction

The R8C Family has a number of peripherals suitable for motor inverter drive applications. In this application note we will take a look at the R8C/24 Group, specifically the R8C/25 driving a 3-phase BLDC motor using the 120° trapezoidal method, also referred to as 6-step. The 6-step method is one of the simplest method for driving 3-phase BLDC motors and in the past was done using discrete logic gates, but with the more powerful peripherals available in today's microcontrollers, such as the R8C Family, we can provide more functionality, better energy usage, and higher safety level when driving motors.
In this application note we will show: 6-step commutation using HALL Sensors, speed measurement using TimerRB, Current measurement using A/D and a basic Proportional-Integral (PI) Speed loop.

This application note is intended to be a primer in 6-step commutation and using the R8C peripherals to drive a three phase BLDC motor. It is not intended to provide in-depth theoretical motor training.

3.1. Basic 3-phase Inverter Topology.

3.1.1. General Inverter Topology

Figure 3.1 shows the basic Inverter topology for driving 3-phase motors. In the case of this application note, the control electronics would be the R8C/25. It is important to note that it requires 6 outputs to drive the inverter section. Timer RD in the R8C family is perfectly suited for this task, and supports "dead-time" to prevent "shoot-through" current on the IGBTs.

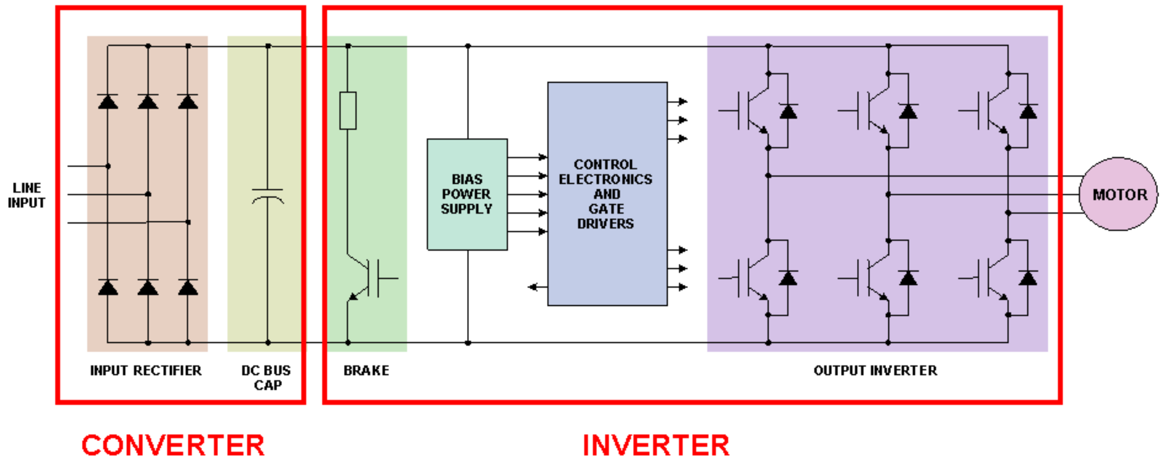


Figure 3.1: Basic Converter/Inverter Connections

3.1.2. R8C/25 Specific Inverter Topology

Figure 3.2 shows the specific architecture of the YMCRPR8C25 Demo board used to develop this Application Note. Note the additional functions over the basic inverter motor drive topology. We can use A/D channels to monitor the Bus voltage for "droop", motor current for excessive current or to implement "torque control". We can use the R8C timer set to measure speed and implement a speed control loop rather than just the simple open-loop control.

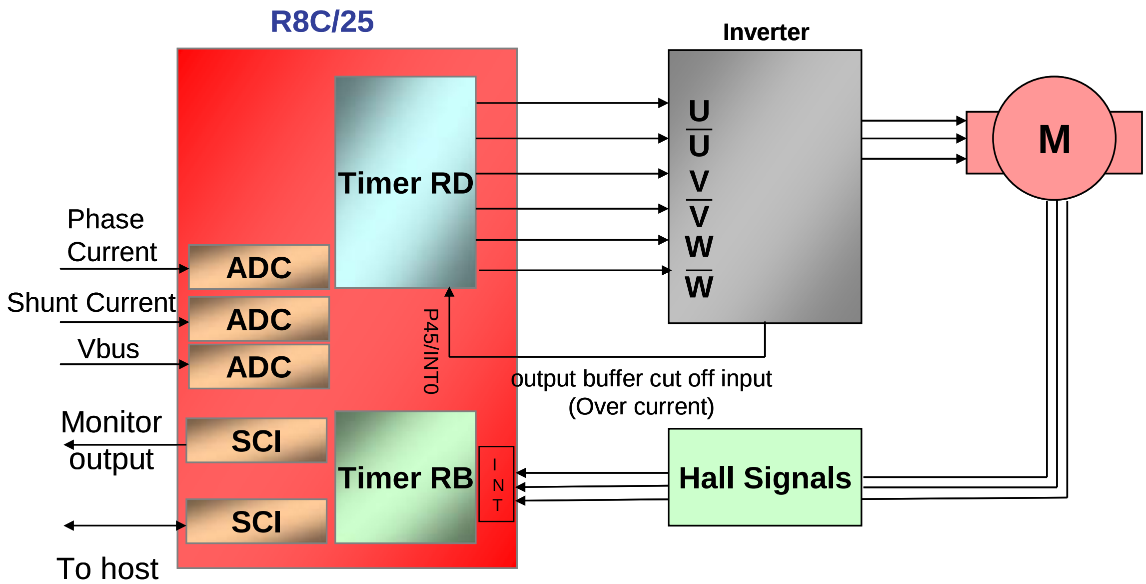


Figure 3.2: Block Diagram, R8C/25 Motor Control

3.2. Basics of Trapezoidal Commutation (6-step).

The 6-step method is one of the simplest methods for driving 3-phase BLDC motors. It is also know as 120-Degree trapezoidal, since it drives each winding for 120-degrees of the electrical rotation and leaves the winding un-driven for 60 degrees. Note, although the drive method is simple, this lack of drive for 60 degrees also results in higher torque ripple in the end application. The system designer must decide if this is acceptable or other drive methods should be considered.

3.2.1. Controlling Phase Voltage

The basic voltage control for the three windings of the motor is performed using Phase-Width Modulation (PWM). In section 3.1 we showed how we will connect the microcontroller to the power inverter stage to control the gates of the IGBTs. In effect, the PWM duty cycle controls the voltage at the motors terminal. There are various modulation methods used in today's inverter drives. Typically modulation techiques are Upper modulation, Lower Modulation, rotating Modulation, or Balanced Modulation. For this application note we will be showing Upper modulation only. Figure 3.3 shows the basic upper modulation waveforms. Note that in this method, only the "P" or upper IGBTs are modulated.

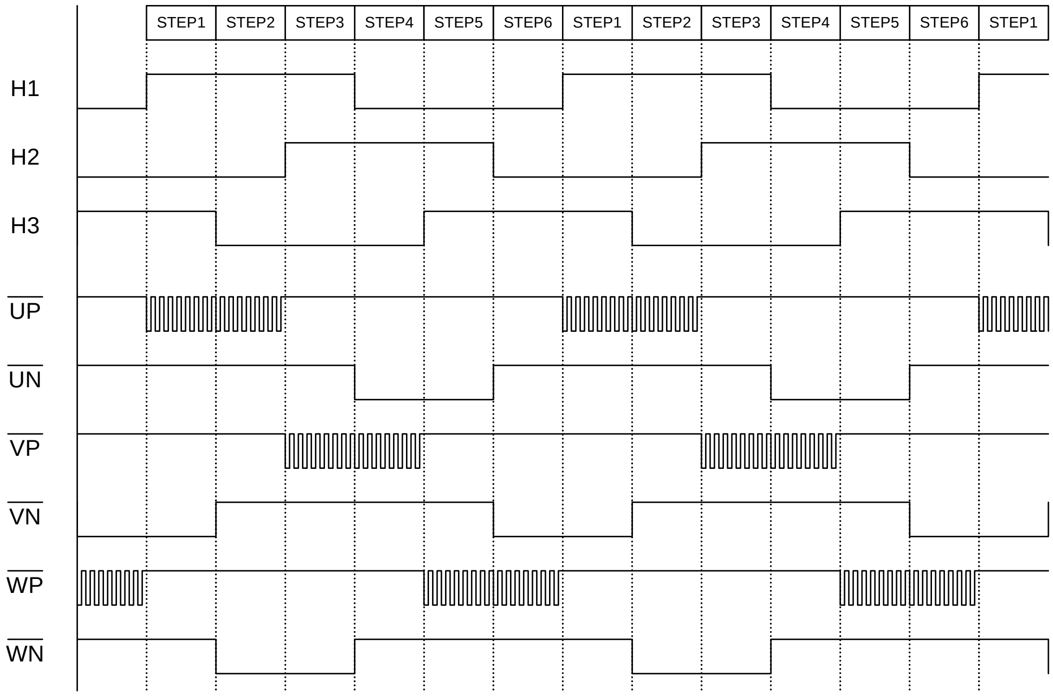
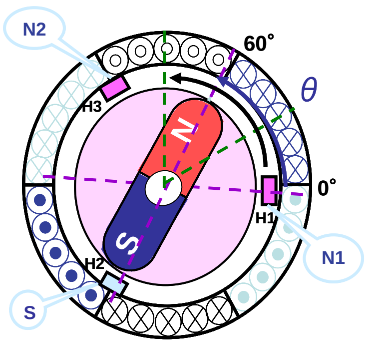


FIGURE 3.3: Upper Modulation (Active-low Drive)

NOTE: Timer RD in the R8C Family can do any of the modulation techniques and is not limited to single-sided modulation techniques. This would include sinusoidal modulation for other motor types such as 3-phase induction.

3.2.2. Rotor Position

Now we have control over the voltage on the windings (and indirectly the current) through the use of a PWM timer, but we must present these signals in the appropriate sequence to properly commutate the motor. In order to do this we must know the rotor position. We will do this with the Hall sensors which sense the position of the rotor. They can do this because they are positioned relative to each motor phase winding in the stator coils (see figure Figure 3.). Figure 3.5 shows a typical HALL cell signature. Note that the state changes every 60° for one electrical cycle. We can then read these on GPIO pins of the R8C and decode them into a 60° rotor position. The number of electrical cycles in one mechanical rotation is based on the number of "pole-pairs" (magnetic poles) in the motor. For the figure given this is 1 pole pair, the motor in the YMCRPR8C25 demo kit has 5 pole-pairs.



H1, H2 and H3 are Hall IC

Figure 3.4: Hall Mounting

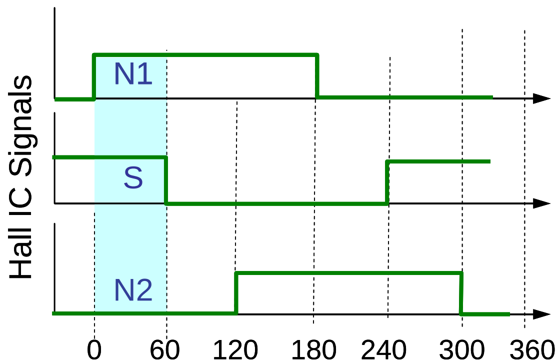


Figure 3.5: Typical HALL Signature

3.2.3. Commutating

So now we can control the voltage, we know where the rotor is so let's put them together.
A motor manufacturer's data sheet will typically tell you *"…when you see this HALL signature, drive these phase windings"*. Figure 3.6 shows a typical commutation sequence. For this motor, when we see the HALL signature for STEP1, we drive U_P and V_N. The rotor will move because the torque being caused by the magnetic fields in the stator coils are being applied at the correct angle to the magnets on the rotor and they will attempt to align. When we see the Hall signature change state to indicate Step 2 we switch the drive from U_P and V_N to U_P and W_N, and the rotor will continue to move. This will continue for the entire cycle until we are back at Step one and the process repeats. This is how we commutate the motor; the speed it rotates at is a function of the motor current which we set by controlling the phase voltage as outlined in section 3.2.1.

■ = Transistor Active

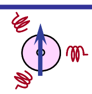
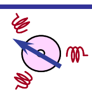
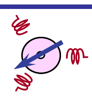
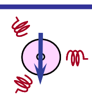
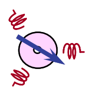
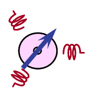
Step 1	**Up**	Vp	Wp	
	Un	**Vn**	Wn	
step 2	**Up**	Vp	Wp	
	Un	Vn	**Wn**	
step 3	Up	**Vp**	Wp	
	Un	Vn	**Wn**	
step 4	Up	**Vp**	Wp	
	Un	Vn	Wn	
step 5	Up	Vp	**Wp**	
	Un	Vn	Wn	
step 6	Up	Vp	**Wp**	
	Un	**Vn**	Wn	

Figure 3.6: Typical Commutation sequence.

3.3. Measuring Speed

There are a few basic methods for measuring the mechanical speed in a rotating motor. The simplest is to use a tachometer attached to the rotor that outputs a given number of pulses for a single rotation. In essence, we will use the Hall Sensors as a tachometer of sorts. Although not attached to the rotor, they do give a fixed number of pulses for a single rotation (i.e. a direct relationship to rotor speed). We will feed the Hall sensors into an interrupt pin of the R8C and detect both edges to double the number of "speed measurements" in a given rotation. This number will be: Number of Hall sensors * number of edges * the number of pole pairs in the motor = measurement points. For the demo motor which has 5 poles, this will be: 3 * 2 * 5 = 30 measurements per rotation. If we measure the time between each interrupt using a fixed processor clock and timer, we can then derive the speed.
Speed = clock period * number of counts.
We have in effect created an input capture timer using three inputs.

The code uses Timer RB as a 16 bit down counter. It does this by reading both the pre-scaler and the Timer RB register during the interrupt service routine. The 16 bits are kept coherent by stopping the timer to prevent a pre-scaler underflow in between the read of the pre-scaler and the read of the timer. The value is assembled in a structure accessible as 2 BYTEs or a WORD. The timer and pre-scaler are then reloaded with the normal reload value (0xFF and 0xFF).

Since the upper level code that converts counts to RPM uses positive numbers, the value is inverted by subtracting it from the timer RB starting count of 65,535 (0xffff) and this value is returned.

In applications that may require very high accuracy, the return value may be adjusted by the number of counts required to stop, read and re-start Timer RB. This value may be derived empirically or determined by evaluating the code using the cycle accurate simulator in the tool chain.

So let's look at an example:
Assume we are using a 2.5MHz clock for timer RB, and we get a count of 2500.
2500 counts / 2.5M counts/second = 1 millisecond.

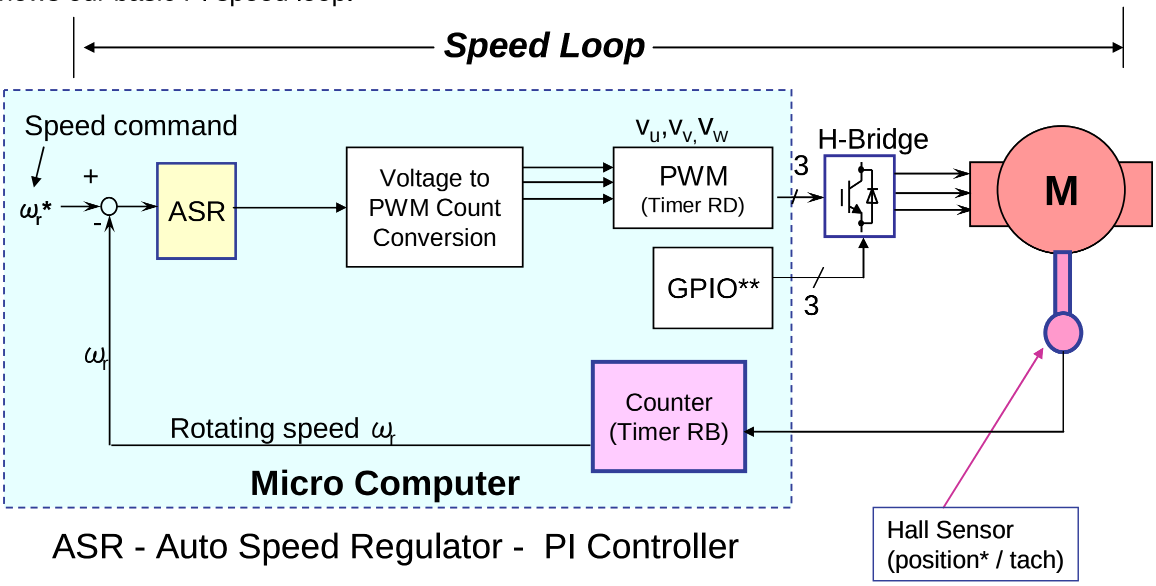
Since this represents 60 electrical degrees or 1/30 of a revolution we can then calculate out:
30 * 1mS = 30mS per revolution or 33.33 Revs per second * 60 seconds/minute = 2000 RPM

In addition to measuring speed, the Timer RB underflow can be used to detect slow or stalled motor.

NOTE: Due to asymmetry in the output of the Hall sensors, it is highly recommended that speed measurements be averaged or filtered. Since the speed loop is typically running at some lower rate than the PWM, this is (typically) not a problem. In addition, if the averaging is a power of 2 (2, 4, 8, etc…) the averaging then becomes just a shift which is easily performed by the R8C.

3.4. Basics of PI Speed Loop

So now we can commutate to move the motor, measure its speed, and vary the phase voltage to control its speed, how do we keep the speed accurate under varying loads? For this we will incorporate a basic PI speed loop, a feedback system (quick run, Control Systems 101 rears its ugly head). Figure 3.7 shows our basic PI speed loop.



* Although it is actually telling us position, we only use that information for commutation
** Not part of control loop, but shown for clarity

ω_r* - Commanded speed in Radians per second
ω_r - Measured speed in Radians per second

Figure 3.7: Basic PI Speed loop

Remember from your basic control systems, a typical loop consists of:

a) feedback element, in this case the Hall sensor and timer RB feeding back speed,
b) error detection, typically summing junction, in this case single line of code that compares commanded speed against measured speed to calculate error
c) Gain stage (amplifier), in this case software that scales the error by gain and computes voltage required to correct the speed.

Very similar to controls you see in hardware.
One important point to note, we can control the voltage of the motor through PWM. Since torque in a motor is related to current and *not voltage*, the output of the ASR calculations is a current reference for the given error. We will need some calculations to convert current to voltage.

So a quick review of Proportional Gain and Integral Gain is in order now.

The most basic type of control loop is **Proportional**, where the **correction** provided is simply a **factor** of how much difference there is between the commanded and actual values. The more error, the more correction. This type is usually stable, but allows some error to exist, so it is **not** good for **highest accuracy**.

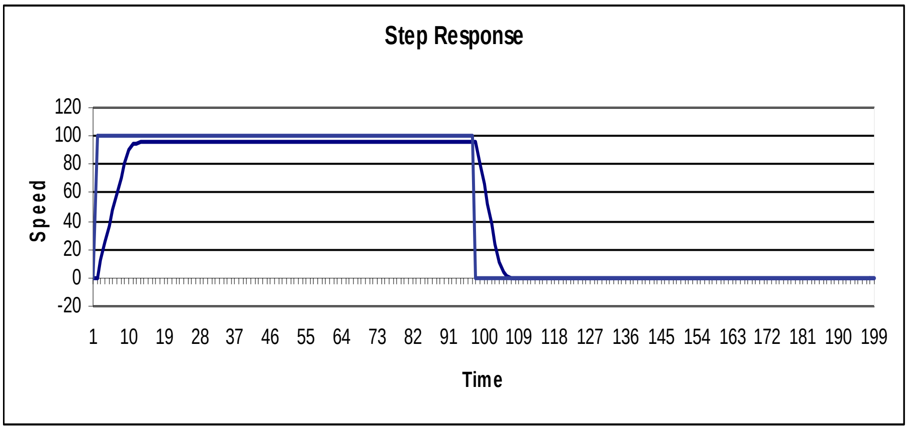


Figure 3.8: Loop Response P-Gain Only

By using an **integrator** in the loop, (adding the accumulated error over time), we can **eliminate long term error** that the Proportional loop allowed. This **improves** the basic **accuracy**. But, these systems can also become unstable and **oscillate** around the set point. Some damping in the system, like friction or resistance, does reduce the tendency to oscillate.

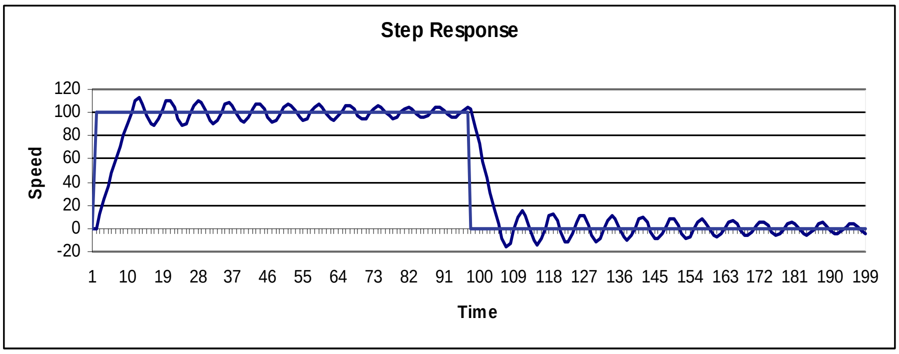


Figure 3.8: Loop Response P-Gain and I-Gain

So now back to our implementation.

So the pseudo-code would look like this:

```
Error = ωr* - ωr;                                          // calculate the speed error
Current_Reference = P-Gain * Error;                        // calculate our current reference
Current_Reference = Current_Reference + Current_Integral;  // add in the integral
Current_Integral = Current_Integral + (Error * I-Gain);     // accumulate the error as Integral
```

The key to this implementation is the fact that the speed loop corrections are applied synchronously to the PWM frequency at some fixed rate that is a sub-frequency of the carrier. So although the speed measurements are asynchronous in nature (i.e. we don't know when a Hall interrupt will come in), we apply the correction at a give frequency.

Also an important point to remember, there are multiple ways to implement these control loops. The way we show it here is very simple in software, but it does require the Integral portion for the loop to converge on the correct speed. If integral gain is set to 0, this loop will tend to "bang" back and forth trying to correct the speed, but never actually hitting the correct speed. How fast it does this is based on the having the correct P-Gain and I-Gain in the software (too in-depth to explore in this application note).

So now we have a current reference representing how we should drive the motor, so we need to get that into the correct “domain” so we can drive the motor with voltage. This means we need to go from Current to Volts (Ohm’s law) to get an absolute motor voltage. After this we need to bring that into our Bus Voltage domain (for example we might need 12V on the motor and we have 24V on the Bus). Finally, the ratio is converted to an absolute count for the PWM timer. This transition is shown in a few lines of pseudo-code below.

```
Voltage_PWM = Current_Reference * Motor_Impedance; // Current to Absolute Motor Voltage
Counts = Voltage_PWM * VBus_Inverse;                // ratio of Motor Voltage to Bus Voltage
PWM_Counts = Counts * PWM_MAX;                      // finally actual PWM counts to load in timer
```

The reader should download the source code as listed in section 5 for complete details and actual code implementation.

NOTE: This sample project for driving BLDC motors is not optimized, rather it is written for clarity an example purposes. Many of the calculations can be rolled into a single calculation. For example a single multiply in the speed loop can encompass both the P-Gain and the Motor Impedance so you get a number back from the speed control function that can be converted to PWM counts directly.

3.5. Implementation and Testing

So now that we’ve given you a basic overview, let’s discuss the quick testing method we did and some insight into how this was implemented in software.

3.5.1. The 6-step state machine and Commutation

So we first had to implement a simple finite state machine so we could commutate the motor and know which windings to drive. We chose to do simple enumerations of the steps and start the numbering at 0 so we can use STEP1 though STEP6 as look-up variable into tables. Since we were making this portable we included some steps (underflow and align) required for “open-loop operation”, but that is for another app note. For now, we can just understand we will use STEP1 through STEP6 (NO_STEP is used to force *no commutation/drive* in open-loop).

SO you can see from the code piece in Figure 3.10, we have defined the steps in such a manner as to provide a direction look-up into the drive tables (i.e. STEP1 through STEP6 equal 0 through 5 respectively). In addition, we have defined an Underflow value to detect when our open-loop commutation should move from STEP1 to STEP6.

Now that these values are set up in this manner we can use them to directly look-up the proper pins to drive in the look-up table. See figure 3.11.

```
typedef enum {
    STEP_UNDERFLOW = -1,
    STEP1,
    STEP2,
    STEP3,
    STEP4,
    STEP5,
    STEP6,
    ALIGN_STEP,
    NO_STEP
} STEP_STATE;
```

FIGURE 3.10: STEP Enumerations

```
#pragma rom high_side_step_table
const UI08 high_side_step_table[8] = {
            UP_ON,   // Step 1, Up Active (modulated)
            UP_ON,   // Step 2, Up Active (modulated)
            VP_ON,   // Step 3, Vp Active (modulated)
            VP_ON,   // Step 4, Vp Active (modulated)
            WP_ON,   // Step 5, Wp Active (modulated)
            WP_ON,   // Step 6, Wp Active (modulated)
            UP_ON,   // Align,  Up Active (modulated)
            0        // NO Step
};
```

```
#pragma rom low_side_step_table
const UI08 low_side_step_table[8] = {
            VN_ON,   // Step 1
            WN_ON,   // Step 2
            WN_ON,   // Step 3
            UN_ON,   // Step 4
            UN_ON,   // Step 5
            VN_ON,   // Step 6
            ALIGN_MASK,       // Align
            0        // NO Step
};
```

FIGURE 3.11: Pin Drive Look-up tables

So you can see when using these tables, if when we see STEP1 on the HALL Cells, we will drive UP_ON on the high-side and VN_ON on the low-side. All we need to do is construct these tables properly for a given motor and connections and it will commutate properly.

In order to test these, *we do not hook them up to the motor or power stage* (bad things happen when motor software has bugs). Rather we hook a signal generator into one of the Hall inputs on the microcontroller to simulate a spinning motor. We then look at our “commutation” using scopes and analyzers. Figure 3.12 shows the actual PWM signals from the processor during our 6-step commutation.

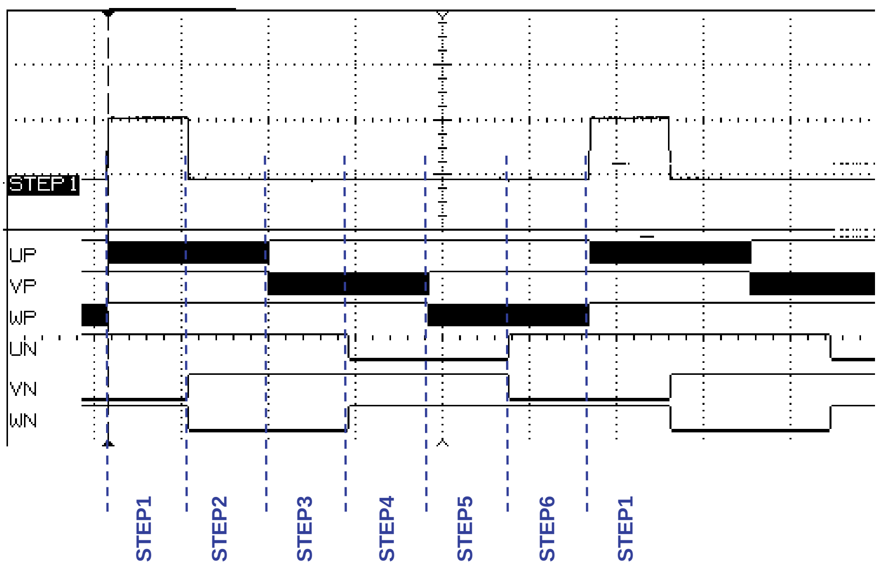


Figure 3.12: Logic analyzer capture of 6-step PWM signals (Active-low).

NOTE: STEP1 signal is I/O port set in software to signal state is STEP1

So looking at these signals, we have a pretty good idea that our motor will spin if driven by this code, *assuming* our Hall Sensor Signature (look-up table) is correct.

3.5.2. Hall Signature Testing

So we still do not need to hook up the motor windings. To reiterate, bad things happen when motor software has bugs! We just hook up the Hall sensor to our inputs and capture the Hall signals with a scope and the software.

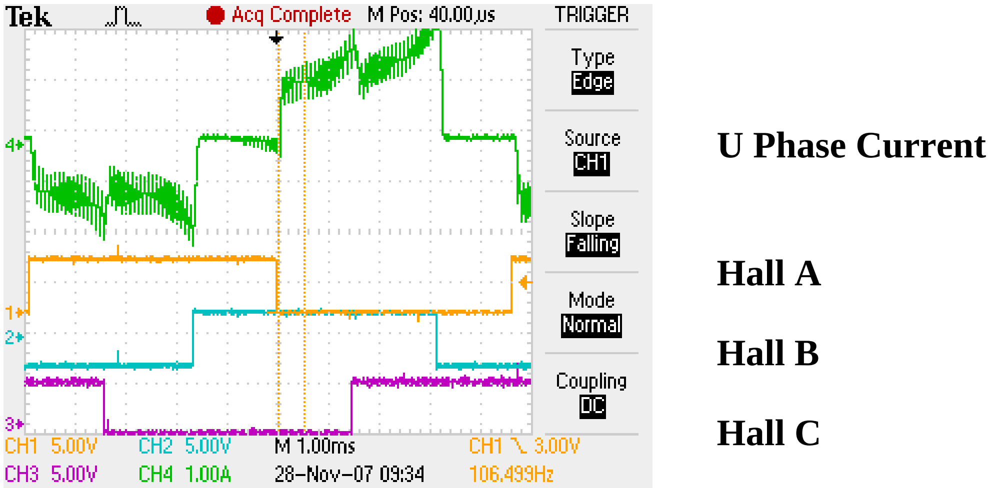


Figure 3.13: Hall Sensor signature and Phase Current

You can see from this figure that the HALL pattern is not directly correlated to a STEP number; however, the software needs to know where the rotor is. For this we construct a revesre look-up table that translates

```
#pragma rom HallTable[]
const HALL_TBL_ENTRY HallTable[] = {
        NO_STEP,0,              // 0 = NO_STEP
        STEP2,   THETA_60DEG,   // 1 = STEP2
        STEP4,   THETA_180DEG,  // 2 = STEP4
        STEP3,   THETA_120DEG,  // 3 = STEP3
        STEP6,   THETA_300DEG,  // 4 = STEP6
        STEP1,   THETA_0DEG,    // 5 = STEP1
        STEP5,   THETA_240DEG,  // 6 = STEP5
        NO_STEP,0               // 7 = NO_STEP
 };
```

FIGURE 3.14: HALL Reverse look-up Table

NOTE: This scope picture was taken from an electrically driven motor, but the Hall signals would look the same if you were spinning it by hand. You can observe the “un-driven” times of the phase current. This is one thing that produces torque ripple in 6-step motor drive.

So electrically they look like we would expect, so how does our Hall look-up table function? For this we need to capture the values in software. As our STEP machine was “commutating” from the Hall sensors, the software tracked where it thought it was and where the Hall sensor told us it was. It captured this in an array named test_data.

NOTE: Since we used enumerations for the steps, the watch window is able to decode and display in basically an easy to read “English” language rather than us trying to decode a 0x05 for example.

3.5.3. Motor Testing

At this point I would feel somewhat safe in hooking up my motor to my power stage and trying to run the motor. My first attempts would probably be with fixed duty cycles on the PWM so as not to worry about "un-tuned PI loop", but it should run.

So finally, let's look at the Motor phase voltages of a properly commutating BLDC motor. You can clearly see the commutation states as shown in figure 3.15. In Step1 we drive U_P and V_N, in Step 2 we switch the drive from U_P and V_N to U_P and W_N, and so on. As an exercise, the reader can validate the rest of the states to confirm the drive matches the drive given in the previously given tables.

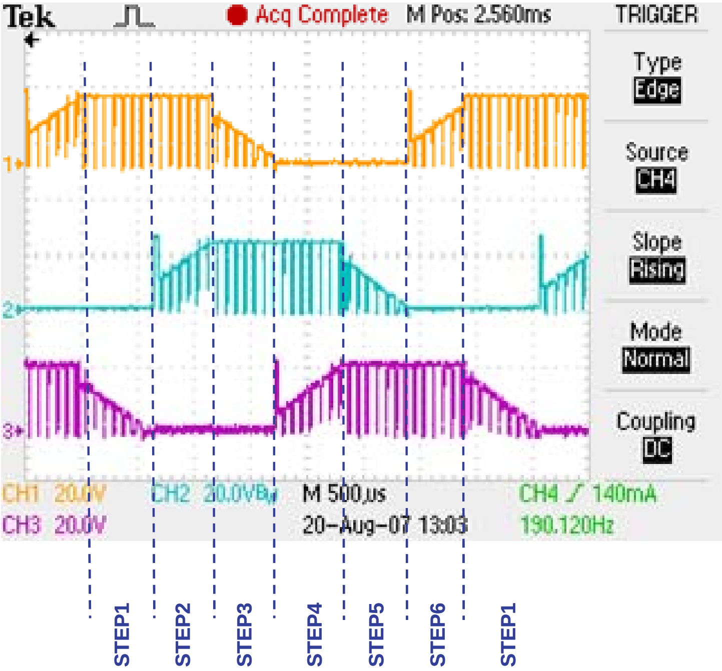


Figure 3.15: Motor Winding on Commutating Motor

.....

4.0 Reference

YMCRPR8C25 Kit
User Manual

Application Notes

REJ05B0845-0100/Rev.1.00	Timer RD in Complementary PWM Mode
REJ05B0486-0100/Rev.1.00	Solutions for Three-Phase Motor Control Programming
REU05B0073-0100/Rev.1.00	Six Step Trapezoidal Control of a BLDC Motor Using Back EMF

Hardware Manual
R8C/24 group, R8C/25 Group Hardware Manual Rev. 1.0
YMCRPR8C25 Motor Control Demo Kit
(Use the latest version on the home page: http://www.renesas.com)

Renesas Technology Corporation Semiconductor Home Page

http://www.renesas.com/

Technical Contact Details:

Global: csc@renesas.com

Inquiries

http://www.renesas.com/inquiry

5.0 Programming Code

The example program was written to run on the Renesas R8C/25 Motor Control Platform (YMCRPR8C25) but could be modified to implement motor control in a user application. The program is written in C (Renesas M16C Standard tool chain V5.43).

Code may be downloaded from the Application Section of the Renesas Website
Filename: an_reu05b0074_r8c_apl.zip
(Use the latest version on the home page: http://www.renesas.com)

6.0 Glossary

The following terms, acronyms and/or abbreviations appear in this app note.

6.1. Acronyms

BLDC – Brushless DC Motor
IGBT – Insulated Gate Bipolar Transistor
PWM – Pulse Width Modulation

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Feb.05.08	—	First edition issued
1.01	Dec.08.08	14	Updated references