Three-phase Motor Control in a Real-Time Embedded Systems Programming Course

Joseph Clifton
Computer Science and Software Engineering
University of Wisconsin – Platteville
Platteville, WI 53818
clifton@uwplatt.edu

Abstract

The Computer Science and Software Engineering department at the University of Wisconsin – Platteville has offered a course entitled Real-Time Embedded Systems Programming since 1999. The focus of the course is on software development for real-time embedded systems and provides considerable hands-on experience. The course covers different-sized platforms and both high-level and low–level aspects of real-time embedded systems programming.

The course has evolved over the years. In 2007, we added closed-loop Proportional-Integral-Derivative control to the course. In 2009, we started using a real motor for the lab instead of a simulated device. The topics of PID control, three-phase motors and commutation, Hall sensors, pulse-width modulation, and the Micrium real-time operating system are covered from a programmer's point of view. This paper describes the course in general and the specifics about the three-phase motor control coverage and project.

1. Background

“Most computers exist not on the desktop, but embedded in other devices. The computers in embedded systems can vary from tiny microcontrollers with a small amount of programming, as found in “low-end” toasters, to big computers running millions of lines of code, as found in large switching systems. As such, the issues in development of software for embedded systems vary greatly. Computer scientists, computer engineers, electrical engineers, and software engineers all have a somewhat different view as to what constitutes a course in real-time embedded systems. Such courses in electrical and computer engineering departments usually emphasize the hardware aspects of embedded systems. Software development is at best a secondary concern and usually done in assembly language. On the other hand, computer science departments that offer such a course tend to focus on the real-time theoretical aspects, in many cases to the exclusion of hardware.” [4]

1.1 Real-Time Embedded Systems Programming Course

The Computer Science and Software Engineering department at the University of Wisconsin – Platteville has offered a course entitled Real-Time Embedded Systems Programming since 1999. The name is meant to convey the fact that the emphasis is on software development, not on hardware development; however, the name is somewhat deceiving since much more than just "programming" is covered. The course is taught at the senior-level with the following pre-requisites:

- CS 2630 - Object-Oriented Programming and Data Structures II,
- SE 3430 - Object-Oriented Analysis and Design, and either
- CS 3230 - Computer Architecture and Operating Systems, or
- EE 3780 - Introduction to Microprocessors

These courses in turn have pre-requisites, so students will have completed several Software Engineering and Computer Science courses prior to taking this course.

The students are exposed to a wide range of topics associated with real-time embedded software development. There is a significant hands-on laboratory experience. The students are expected to use the analysis, design, implementation, and testing techniques learned in previous courses. Students work in small groups for three or four of the projects. For those projects, students are required to use version control and log their time.

1.2 Platforms

One aspect of the course that distinguishes it from other such courses is the "evolution" approach to the platforms on which the students are expected to develop. There are five or six laboratory projects and a final project. Students start by using a very small PIC microcontroller (1K flash, 64 bytes RAM). They program it in assembly language. Then they are given a C compiler, a bigger PIC (8K flash, 368 bytes RAM) and significantly more challenging programs to develop. The students spend about six or seven weeks with the PIC, doing four laboratory projects. After that, they switch to a more “sophisticated” microprocessor: faster, more RAM, more flash, more peripherals. From 1999 – 2009, an 8051-based derivative was used. For the past two years, a Freescale microcontroller has been used. In all cases, they are required to do a large-sized multi-tasking program using a small Real-Time Operating System (RTOS).

In 2007, at the suggestion of members of our Advisory Board, we added closed-loop Proportional-Integral-Derivative (PID) control to the course. The students were assigned a project involving an 8051-derivative microprocessor, Keil C, and Keil’s RTX 51 Tiny real-time operating system. The project involved multiple tasks, but the main thrust was control of a simulated analog device. The students were given about a month to complete the design, test specification and implementation of this lab.

In 2009, we started using a real motor for the lab instead of a simulated device. We switched to a Freescale MC56F8037 digital signal controller, the Micrium uC/OS-II real-time operating system (RTOS), and Freescale’s CodeWarrior. For the motor, we chose the Freescale 56F8000 Motor Control Board. It has a Maxon EC-200187 brushless three-phase motor and Hall-Effect sensors. It has a daughter card connector to the MC56F8037EVM Evaluation Board.

From 1999 – 2003, the students used 80386 boards for the final project. From 2004 – 2010, the students used an XScale processor with touch screen running the Windows CE operating system and the .NET Compact Framework. The students were required to do a real-time UML design using Rational Rose and develop in C# using Microsoft Visual Studio .NET. Some sample final projects include an ATM, RFID cash register, milking parlor control system, sign language interpreter, automatic bowling score keeping system, and sales and inventory tracking system. There were multiple delivery dates for the last laboratory project and the final project, and these dates were interspersed over the last eight weeks of the semester.

In 2011, the final project was changed to require more applied science, specifically physics. This change came about due to ABET (formerly known as the Accreditation Board for Engineering and Technology) assessment of the Software Engineering program and the need to show that students can apply basic science to the design of software. The microprocessor used was a PIC. The student’s application of physics didn’t go quite as well as anticipated, so at the time this paper is being written, it isn’t clear what form the final project will take this spring. However, this paper focuses on the Freescale-based motor controller projects that are laboratory projects five and six.

1.3 Topics Covered

There is a wide range of topics covered in the course. The topics roughly fall into three categories: Overview, Practical, and Theoretical. [4]

The "overview" topics are those that generally include a listing of instances, brief discussions, comparisons and contrasts. The students get laboratory experience only in a small number of the instances of a given topic. For example, the students will only use two or three different microprocessors. The course gives an overview of

- Processors and systems
- Development languages
- Development environments
- Platforms and platform standards
- Real-time operating systems and tasking shells

The "practical" topics are those for which an overview is also given; however, the students get more comprehensive laboratory experience. The practical topics generally include hands-on experience and considerable experience with the hardware. It should be noted that this is not done at an electrical or computer engineering level. The practical topics include:

- Simulators, emulators, target debuggers
- RAM, flash, EEPROM
- Timers, counters, interrupts, watchdogs
- Digtial I/O
- UARTs, SPIs, PPIs
- Hall sensors
- Pulse-Width Modulation (PWM)
- Interfacing and communications

Theoretical topics are those that are primarily software in nature and are generally hardware-independent. These topics allow traditional computer science material to be applied to the unique problems of development of software for real-time systems. Except for the last one, such topics as these that can be found in a text like [3]. The theoretical topics include:

- Reliability, fault tolerance, exception handling
- Concurrent programming, tasks, threads
- Synchronization and communication

- Resource control: semaphores, monitors
- Process and resource scheduling
- Device and inter-processor communications
- Real-time UML and object-oriented design of embedded systems

In addition to lecture topics, each student is required to give a concise 15-minute presentation on a topic related to real-time embedded systems programming. The students are allowed to choose from a list of over 25 topics or choose a pre-approved topic of their own. The list includes topics such as ARINC, Bluetooth, CAN, CRCs, embedded internet, FAA standards, fixed-point arithmetic, GPS, specific RTOS's, and safety engineering.

2. Three-phase Motor Control

From an electrical and hardware engineering perspective, motor control theory is typically covered in a semester-long course requiring several electrical engineering courses as prerequisites. For us to take such an approach is clearly infeasible. Therefore, we distilled down the essentials needed from a programming point-of-view to:

- Software closed-loop control via Proportional-Integral-Derivative (PID) control
- Speed and position sensing via Hall sensors
- Three-phase motor commutation
- Pulse-width modulation

These topics in addition to the Micrium RTOS are covered from a programmer's point of view in less than two weeks.

2.1 Proportional-Integral-Derivative Control

A PID controller has a measured value and a desired value as inputs. The algorithm calculates the instantaneous error, which is the difference between the desired value and the measured value. For example, if the desired speed is 8000 revolutions per minute (RPM) and the measured speed is 8500 RPM, the instantaneous error is -500 RPM. The algorithm then uses the instantaneous error as well as the sum and differences of errors to calculate a control output value. The output could be a value that is directly used to control a device, or could be a correction value to apply. For example, the output could be the actual Pulse-Width Modulation (PWM) value to apply to run a motor at a desired speed, or it could be the amount to change the current PWM value.

The PID software algorithm generally runs at a constant rate to simplify the time component of the calculations and allow them to be wrapped up in the PID constants. The output typically uses one or more of the following calculations:

- Proportional – constant times instantaneous error
- Integral – constant times sum of instantaneous errors
- Derivative – constant times the change in error

The proportional term is the simplest of the PID terms and attempts to fix the error using the difference in the desired value and actual value. If only a proportional term is used, a small constant results in a long delay in achieving the desired value. On the other hand, if a large constant is chosen, there is a tendency to overshoot the desired value. A strong overshoot that continues to travel back and forth is known as ringing. [6]

The integration term can be thought of as the "history" of the error and is used to smooth out the error over time. The integration term grows larger the longer the actual value stays on one side of the desired value. It starts to decrease once the actual value goes to the other side of the desired value and helps diminish the proportional ringing effect mentioned above. One problem with this term is that in some cases, it can grow quite large, even unbounded. Therefore, this term is typically restricted to an appropriate range based on the properties of the item being controlled.

The derivative term gives the instantaneous rate of change of error. It can be thought of as a "prediction" of future error since it is the "slope" of the error function. As the actual value gets closer to the desired value from the same side, the proportional and integral terms will have the same sign. However, the derivative term will have the opposite sign since the instantaneous errors will be getting smaller. For example, if the previous error was 10 and the current error is 5, the derivative term would be a constant times -5 whereas the proportional and integral terms would both be positive. It should be noted that since the derivative term is a rate of change, it should have a denominator of a delta time. If we assume the PID loop running at a constant rate, then the delta time is constant and can be wrapped up in the derivative term constant.

Assuming that the PID loop is executed at a constant rate, the PID control algorithm is surprisingly simple:

```
currentError = desiredValue – currentValue
pTerm = Kp * currentError
iSum = iSum + currentError
if  iSum > iSumMax then
   iSum = iSumMax
elsif  iSum < iSumMin then
   iSum = iSumMin
iTerm = Ki * iSum
dTerm = Kd * (currentError – previousError)
previousError = currentError
controlOutput = pTerm + iTerm + dTerm
```

The developer is responsible for choosing the three PID constants Kp, Ki, and Kd, and the integration limiters iSumMax and iSumMin.

Coming up with the constants is referred to as "Tuning the PID loop". "The nice thing about tuning a PID controller is that you don't need to have a good understanding of formal control theory to do a fairly good job of it. About 90% of the closed-loop controller applications in the world do very well indeed with a controller that is only tuned fairly well." [6]

2.2 Three-phase Motor Commutation

The study of motor control encompasses a broad and deep set of knowledge, with varying amounts of formal education required in areas such as Physics (E&M), Electrical Engineering, and Mechanical Engineering. We obviously cannot go into any significant theoretical depth in our course. Instead, a high-level view of the required theory is given, and the emphasis is on the practical details required for motor control. Furthermore, we restrict the coverage to brushless three-phase direct current (DC) motor control with Hall sensors.

A brushless three-phase motor consists of 3M coils and 2N magnetic poles. The left side of Figure 1 shows a motor with three coils (M = 1) and two poles (N = 1). The coils are usually referred to a U, V, and W or A, B, and C, depending on which documentation you read. The magnetic poles refer to the north-south poles of a magnet. An M and/or N greater than one allows control of the motor in smaller steps. For example, a motor with three coils and two pole pairs (four poles), as shown in the right side of Figure 1, requires two complete "electrical" revolutions to achieve one mechanical revolution. Figure 2 shows other combinations of coils and poles. Brushless refers to the fact that the coils are fixed and the magnets rotate, eliminating the problem in a conventional DC motor of connecting current to moving coils. To say it another way, the magnets versus the coils are part of the rotor. For purposes of the coverage in class, these details are mentioned briefly but the primary focus is on the high-level theory of a motor with three coils and one pole pair.

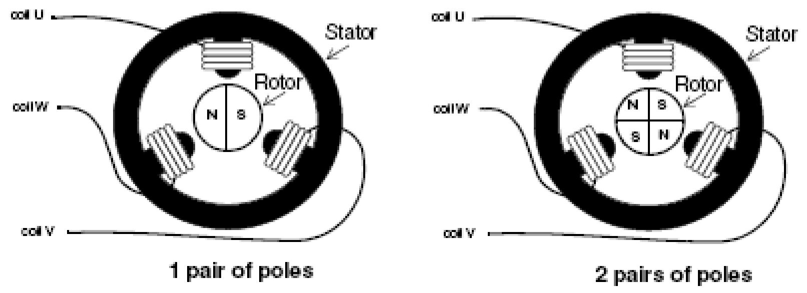

Figure 1 - Three Phase, Three Coil Brushless DC Motor (Figure taken from [1])

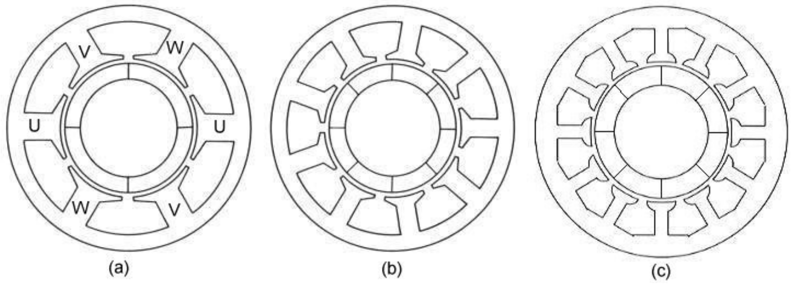

Figure 2 – Different Type Three-Phase Motors: a) 6 coils, 4 poles, b) 9 coils, 8 poles, and c) 12 coils, 8 poles (Figure taken from [2])

For each rotor position in a motor with three coils and one pole pair, movement of the rotor in either a clockwise or a counter-clockwise direction requires that one coil be connected to negative voltage, one to positive voltage, and the third be unconnected (allowed to float). This forces the rotor to move in the desired direction. The direction of the current through the coils determines the generated magnetic field orientation. The magnetic field either attracts or repels the rotor magnets. Therefore, by changing this at the right time and in the right sequence, the motor rotates. Motor commutation refers to the sequence and timing of applying the voltages to move the motor in the desired direction. There are only six different possibilities for energizing the coils to achieve rotation in a given direction, as explained below.

The current position of the rotor can be determined by placing three Hall sensors at a 120-electrical-degree separation. Hall sensors are analog devices, varying their output voltage with the magnetic field. They are based on the Hall Effect, which states that a magnetic field perpendicular to a current through a conductor will produce a voltage differential transverse to the current. For applications such as motor control, extra circuitry is typically added to turn them into discrete devices, with output of zero or one.

For motor control, discrete Hall sensors can be used to detect the polarity of the last magnetic pole that was nearby. They are available with different characteristics. For this paper, it is assumed that when a south magnetic pole is near the Hall sensor, the Hall sensor outputs a one. When a north magnetic pole is near the Hall sensor, it outputs a zero. The output stays the same until the rotor approaches with polarity opposite of the last sensed state.

Many sources explain how Hall sensors can be used to drive commutation, for example, [1,2,5]. Unfortunately, they vary considerably due to sensor placement and polarity. Furthermore, they often do not specify how the sensors are placed. Figure 3 provides one conceptual way to visualize what is required to commute the motor. The coils U, V, and W depicted in Figure 1 above are referred to as A, B, and C in Figure 3. Note that the coils are shown "strung out" (an electrical schematic) versus wrapped as in Figure 1. The rotor is shown as a magnet, with the black half being the south magnetic pole and the white half being north. The diagram also shows the Hall sensor output for various sectors of the "electrical circle". The source for this diagram does not specify how the sensors are placed. However, by studying it and comparing it with other sources, one can deduce that Hall sensor A is placed by coil C, Hall sensor B is placed by coil A, and Hall sensor C is placed by coil B. Also, note that the magnet is portrayed as "long and skinny" whereas it would typically look more like that in Figures 1 and 2.

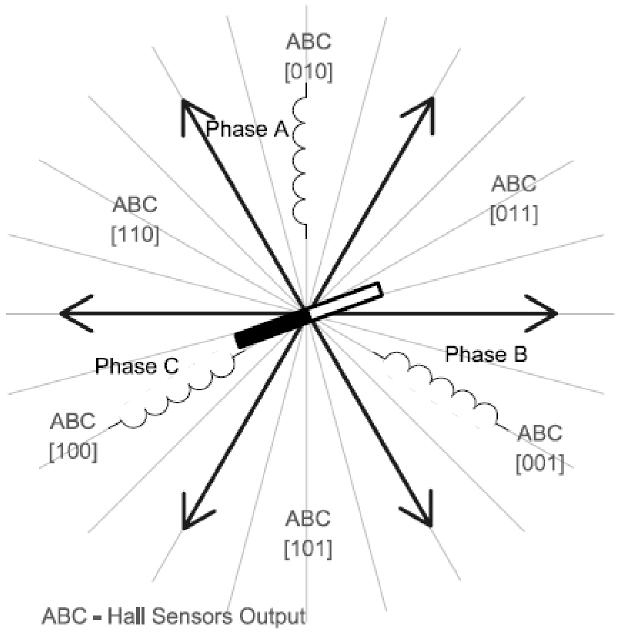

Figure 3 – Hall Sensors and Six-Step Control (Figure taken from [5])

In Figure 3, the south magnetic pole is in the sextant that contains coil C. To derive the Hall sensor value, note that the south pole is near the sensor at coil C, so it would read a one, and the north pole is near the sensors at A and B, so they would read zero. Then recall that Hall sensor A is at coil C, so the Hall sensor output bit pattern is [ABC] = 100. As another example, suppose the rotor moved forward into the next sextant. Then the south pole is near coils A and C, so they will read one and the north pole is still near coil B, so it will read zero. Therefore, the sensor that changes is the one at coil A, which is sensor B. The output bit pattern changes to 110.

Given the rotor position in Figure 3, to make the motor move in a clockwise direction, a positive voltage must be applied to coil A and a negative voltage applied to coil B. The positive voltage at A will repel the north pole and the negative voltage at B will attract the north pole, thus forcing the magnet to move in a clockwise direction. Coil C is left floating. When the rotor reaches the next position (110), coil A is kept positive to attract the south pole, a negative voltage is applied to coil C to repel the south pole, and B is allowed to float.

Assuming the Hall sensors are placed at the coils, all possible commutation tables are obtained by permutations of the sensor placement and sensor polarity. There are only six allowable bit patterns. Table 1 shows the commutation sequence for the sensor placement in Figure 3. Viewing the Hall sensor output bit patterns as integers, the sequence for clockwise commutation is (4, 6, 2, 3, 1, 5) or any cyclic permutation of this. The sequence for counter-clockwise commutation is (4, 5, 1, 3, 2, 6). It should be noted that any rearrangement of the Hall sensors or change in polarity always results in one of these two cycle patterns. For example, flipping the A and B sensors in Figure 3 (100 -> 010, 110 -> 110, etc.) results in the sequence (2, 6, 4, 5, 1, 3) which is a cyclic permutation of the counter-clockwise commutation sequence. An easy way to show that these are the only two possible sequences is to start with the sensor pattern 010. Then one sequence is generated by applying (flip bit 0, flip bit 1, flip bit 2) cyclically and the other by applying (flip bit 2, flip bit 1, flip bit 0) cyclically. Any permutation of the sensors applied to one of these two cyclic patterns will result in one of these two patterns. Thus, by applying one of these two sequences, the motor will always turn either clockwise or counter-clockwise.

Sensor A Positioned at Coil C	Sensor B Positioned at Coil A	Sensor C Positioned at Coil B	Coil A Phase	Coil B Phase	Coil C Phase
1	0	0	+V	-V	Float
1	0	1	Float	-V	+V
0	0	1	-V	Float	+V
0	1	1	-V	+V	Float
0	1	0	Float	+V	-V
1	1	0	+V	Float	-V

Table 1 - Clockwise Commutation Sequence Corresponding to Figure 3

2.3 Speed Control

Motor speed can be controlled using Hall sensors, Pulse Width Modulation (PWM), and a PID loop.

Under no load, the motor speed will vary proportionally with the voltage applied. PWM is a technique to produce variable voltage on a discrete output pin. The pin is switched on and off at a very fast rate. The duty cycle is the percentage of “on time”. For example, suppose a discrete pin outputs five volts when it is on and zero volts when it is off. If the pin is switched on and off at a fast rate such that it is on half the time and off half the time, the duty cycle is 50% and the voltage would be 2.5. If it is on all the time, the duty cycle is 100%. If it is off all the time, the duty cycle is 0%.

The estimation of the current speed is obtained from the Hall sensors. By measuring the rate at which the Hall sensors change and knowing the electrical-to-mechanical ratio, a simple calculation gives the motor speed. For example, suppose a motor has four pole pairs. It takes four electrical revolutions to achieve one mechanical revolution. Associate a timer and interrupt with one of the Hall sensors. Assume the interrupt is configured to trigger on both the rising and falling edge, so it will trigger each time there is a pole change for the selected Hall sensor. There are two pole changes per electrical revolution and therefore, the interrupt will trigger eight times per mechanical revolution. The interrupt handler measures timer "ticks" since the last interrupt. Suppose that there are X timer ticks since the last interrupt. Then an estimation of the current motor speed is given by: Timer_Tick_Frequency / (X ticks per 1/8 mechanical revolution). If the timer runs at 500 kHz, the speed is given by:

(500000 ticks per sec) / (X ticks per 1/8 rev) = (62500 / X) revs per second

Note that in this case, a 16-bit unsigned integer division could be used, truncating to the nearest integer.

Given the desired speed and the estimate of the current speed, a PID loop is used to determine the PWM value to apply. Two approaches can be taken. One is to choose the PID constants such that the output of the PID loop is the PWM duty cycle to apply to the coils. In this case, there must always be at least some error, otherwise the PID output, and hence the PWM duty cycle, would be zero and therefore the PWM voltage would be zero. The other is to treat the PID output as a delta (change) to the current PWM duty cycle. In this case, the PID output specifies how much to increase or decrease the current PWM duty cycle to achieve the desired speed.

3. Project Description

The motor control assignment is given as two laboratory assignments. The first laboratory assignment is given to familiarize the students with the MC56F8037 processor, Freescale Code Warrior, and the Micrium uC/OS-II RTOS. The second laboratory assignment is speed control of a three-phase brushless DC motor.

3.1 Introductory Assignment

This lab introduces the students to the environment they will be using for the motor control project. It covers ADC, DAC, GPIO, PWM, and interrupts on the MC56F8037. It covers the Freescale CodeWarrior IDE and the use of its Components Library. In addition, it covers tasks and mailboxes in Micrium uC/OS-II. It is a "quick and dirty" lab. The students are given a week to complete the laboratory and are not required to document their code beyond what they want for their own reference.

The program has three tasks, two interrupts, and two mailboxes. The interrupts are associated with two buttons, which when pressed, place "increment" and "decrement" messages in the first mailbox. One of the tasks removes the messages and modifies the PWM and DAC accordingly. The PWM output drives an LED whose brightness varies with the PWM duty cycle. The DAC is connected to an ADC. One task periodically reads the ADC and places the result in the second mailbox. The third task gets the results from that mailbox and displays the high-order bits on LEDs.

3.2 Motor Control Assignment

The lab is given in the second half of the semester and the students are given about a month to complete the design and implementation. However, that month spans over spring break, so they actually only have about three weeks. Although the programming language used is C, the students are required to do a UML design, and it is expected that OO-type concepts will be used to produce quality code. The lab is structured to require the students to use most of the topics covered in the course.

The students work in groups of two or three. Each group has a Freescale MC56F8037EVM Evaluation Board and a 56F8000 Motor Control daughter board. Students are required to use version control and log time spent on the project (using a homegrown departmental tool) with specifics about what they accomplished during each working session.

The following is an overview of the details given for the program:

1. **RTOS** – Use uC/OS-II and set the timer tick to one millisecond.
2. **Reading Hall Sensors –** The Hall sensors for the motor are on GPIOB pins 2, 4, and 5, with Hall sensor A connected to pin 4, B to pin 5 and C to pin 2.
3. **PWM** - The PWM outputs are set up in complimentary pairs, with one output to one end of the coil and the other to the other end. For example, PWM0 is connected to one end of coil A and PWM1 is connected to the other end. By selecting complimentary pairs, when one end of the pair is high, the other is low and vice versa. When the commutation cycle specifies that coil A should be positive, it means PWM0 should be "on" (high) and PWM1 should be "off" (low). Of course, unless it is 100% duty cycle, it will not be that way for the full period. For some of the period, it will be the other way around. If the duty cycle is 50%, then for half the time, PWM0 is high, PWM1 is low, and for the other half of the time, it is the other way around, which means the rotor will not move. Therefore, to get movement corresponding to a positive voltage on coil A, the duty cycle for PWM0 needs to be greater than 50%. For our motor, there is 9V to coil A when the PWM0 duty cycle is 100%. To determine the voltage to coil A for other PWM values, take 9V * (duty cycle PWM0 – duty cycle PWM1). Since these are complimentary pairs, the duty cycle of PWM1 is 100% - duty cycle of PWM0. Therefore, the voltage is 9V * (2 * PWM0 duty cycle – 100%). Thus, if PWM0 duty cycle is 75%, the voltage to coil A is 9V * (150% - 100%) = 4.5 V.

Set the PWM frequency to 128 kHz. Center-aligned PWM will yield half that value, namely 64 kHz. You could get by with smaller, but this processor is fast enough to handle it.

4. **Commutation** - To move the motor, follow the pattern for setting the coil voltages as specified in class. To do this, use the CodeWarrior generated SwapAndMask function. Mask refers to which channels are on and which are off. There will always be two pairs on and one pair off. Swap refers to whether the coils are connected to positive or negative. No swap corresponds to positive. Swap corresponds to negative and the two PWM outputs of the complementary pair are physically swapped. Thus, the duty cycle does not need to change to move through a rotation. The parameters for the SwapAndMask function are of type TChannelPairs and TChannels. These are defined as structs with one-bit fields. As an example, by looking at the commutation table, if the Hall sensor reading is [100], phase coil A should be positive, phase coil B should be negative, and phase coil C should float. For the Swap setting, pair 0 should not be swapped (set it to 0), pair1 should be swapped (set it to 1), and it does not matter for pair 2 since it will be masked. For the Mask setting, channels 0 – 3 should be set to 0 (not masked) and channels 5, 6 should be set to 1 (masked, which means the PWM output is disconnected, and coil C will float).

   Create a Commute Motor task. It should be higher priority than the other tasks. For its tasking loop, have it wait on a synchronization semaphore that tells it to do a commutation. It reads the Hall sensors to get the Hall state and then calls SwapAndMask to energize the coils. The semaphore will be signaled by the Hall sensor interrupts (described below). Have a timeout of about 10 milliseconds for the wait. That way, if the sensors do not trigger for whatever reason, a commutation can still occur if needed.

5. **Hall Sensors** – Set up edge-triggered interrupts for Hall sensor changes. The interrupts must trigger on both falling and rising edges. In the interrupt handlers, post the commutation semaphore referred to in the Commutation section.

6. **Speed Calculation**: To calculate speed, use a 16-bit timer in one of the Hall sensor interrupts and proceed as discussed in class.

7. **PID Loop Control** – Under no load, the speed is proportional to the PWM voltage. Since 128 KHz was chosen for the PWM, this implies that the Counter Modulo register is 250. This specifies the number of PWM "ticks" per cycle and the duty cycle is the number of ticks that the PWM is "high". Therefore, a PWM value of 0 means that for 0 out of 250 ticks it is high, and for 250 it is low. A PWM value of 250 means it is high for all 250 ticks. However, since the PWMs are set up in complementary pairs, a PWM of 50%, which is a value of 125, means the motor is stopped. Thus, values greater than 125 and less than or equal to 250 are needed to move the motor. Your goal is to control the number of motor revolutions per second. Assume a maximum speed of 200 revolutions per second. Scale so a PWM duty of 125 corresponds to a speed of 0 and a PWM of 250 corresponds to the maximum motor speed. That means the motor speed under no load is given by:

Motor speed under no load = (PWM_Value – 125) * MaxSpeed / 125

Note that for PWM values less than 125, the motor spins the other way, so this is actually a motor velocity. To achieve a desired speed under no load, solve to yield:

PWM_Value = 125 + No_Load_Motor_Speed * 125 / MaxSpeed

However, it is often the case that there will be a load, so you cannot simply apply the formula. You need to determine how to use the PID output to set the PWM. The PID loop manipulates error (desired minus actual) to produce an output, which can be the amount to change the PWM value to achieve the desired speed. The "gain" for the Proportional term, Kp, will be related to the factor above: 125 / MaxSpeed. To tune the loop, take an approach suggested in Wescott, T., "PID Without a PhD": Start with some reasonable values and adjust until you get a desirable result. When testing with a "load", **gently** put your fingers on the turning motor and squeeze, but don't squeeze so hard you stop it or cause an over-current. You are not implementing torque or current control! Create a PID Loop task based on the pseudo-code for the PID loop given in class. It should be lower priority than the Commutation task but higher than the Serial Output task. Run the task at about 10 milliseconds.

8. **Setting Desired Speed** – Use the third pushbutton (S3, pin GPIOB3) to change the desired speed. Start it at 0, then as the button is pressed, cycle the desired speed through 40, 80, 120, 80, 40, 0, etc. If the desired speed is 0, disable the PWM; otherwise, the PID will drive it back and forth around 0. This could be bad, since you are not implementing code to determine the motor direction (i.e., velocity versus speed), so the PID loop could cause the motor to jump around wildly.
9. **Serial Output of Speed**: Set up a serial port output using 19.2K, 8, N, 1. Create a task that runs about every 25 milliseconds and transmits the current value of the motor speed out the serial port.

3.3 Sample Commutation Code

Below is sample code that can be used to commute the motor.

```
typedef struct _CommuteType
{
  TChannelPairs pairs;
  TChannels chans;
} CommuteType;

const CommuteType COMMUTE[] =
{
  // 1=Swap      1=Mask(Not connected)     Hall   Coil
  // A  B  C     A+ A- B+ B- C+ C-         ABC    A  B  C
```

```
    { {0, 0, 0}, { 1, 1, 1, 1, 1, 1 } },  // 000  Invalid
    { {1, 0, 0}, { 0, 0, 1, 1, 0, 0 } },  // 001  -V NC +V
    { {0, 0, 1}, { 1, 1, 0, 0, 0, 0 } },  // 010  NC +V -V
    { {1, 0, 0}, { 0, 0, 0, 0, 1, 1 } },  // 011  -V +V NC
    { {0, 1, 0}, { 0, 0, 0, 0, 1, 1 } },  // 100  +V -V NC
    { {0, 1, 0}, { 1, 1, 0, 0, 0, 0 } },  // 101  NC -V +V
    { {0, 0, 1}, { 0, 0, 1, 1, 0, 0 } },  // 110  +V NC -V
    { {0, 0, 0}, { 1, 1, 1, 1, 1, 1 } }   // 111  Invalid
};
```

To commute the motor each time a Hall sensor interrupt occurs:

```
byte val = GetHallReading();
PWMC1_SwapAndMask( COMMUTE[val].pairs, COMMUTE[val].chans );
```

4. Conclusion

The laboratory project described above is a good way to introduce the students to a variety of topics typical in real-time embedded systems programming projects, including closed-loop control. It has been an evolution of similar projects with the latest change being the use of a real motor instead of a simulated device. The next logical step would be to incorporate current and torque control into the project. After that, the next step would be to create a "real-world" project that uses the motor control concepts.

References

[1] Atmel Application Note, "AVR194: Brushless DC Motor Control using ATmega32M1", available at: http://www.atmel.com/dyn/resources/prod_documents/doc8138.pdf

[2] Atmel Application Note, "AVR443: Sensor-based control of three phase Brushless DC motor", available at http://www.atmel.com/Images/doc2596.pdf

[3] Burns, A. and Wellings, A., *Real-Time Systems and Programming Languages,* Addison-Wesley, 2009.

[4] Clifton, J., "A CS/SE Approach to a Real-Time Embedded Systems Software Development Course", SIGSCE Bulletin, Volume 33 Number 1, March 2001, pp. 278-281.

[5] Freescale Semiconductor Application Note AN1916, "3-Phase BLDC Motor Control with Hall Sensors Using 56800/E Digital Signal Controllers", available at http://www.freescale.com/files/product/doc/AN1916.pdf

[6] Wescott, T., "PID Without a PhD", *Embedded Systems Programming*, October 2000.