LUMINARY MICRO™

Driving a Brushless DC Motor with a Stellaris™ Microcontroller

APPLICATION NOTE

AN01238-01

Legal Disclaimers and Trademark Information

Luminary Micro, Inc.
2499 South Capital of Texas Hwy, Suite A-100
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction 4
BLDC Motor 4
Block Diagram 4
Software 6
  Controlling Motor Rotation with GPIOIntHandler 7
  Controlling Motor Speed with SpeedHandler 8
  Using a Timer and Event Counter for a Software QEI 9
  Using the Stellaris Hardware QEI 10
  Providing a User Interface through the UART 10
Memory Size 11
Schematics 11
Conclusion 17
References 17
Company Information 17
Support Information 18

Introduction

This application note discusses the characteristics of a brushless DC motor (BLDC), describes how to operate a BLDC motor, and details how to use the peripherals found on Stellaris™ microcontrollers to control a BLDC motor. It includes an example application with software and schematics.

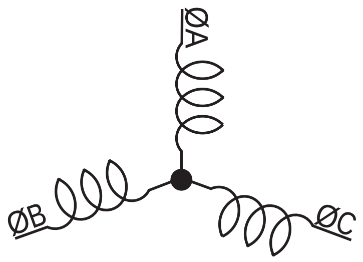
It is assumed that the reader has a good working knowledge of a Stellaris microcontroller and is familiar with the Stellaris Driver Library software. The Stellaris Driver Library software is a set of drivers for accessing the peripherals found on the Stellaris family of microprocessors. The Stellaris Driver Library is available at no cost at http://www.luminarymicro.com.

BLDC Motor

A 3-phase brushless DC motor (BLDC) has three windings, which are typically connected at one end. This connection scheme is known as a wye (Y) or star configuration. Windings may also be connected end-to-end in a delta configuration. The electrical connection points are commonly referred to as phases.

The motor is driven by applying V_{CC} to one phase and GND to another, causing current to flow into the motor through one winding and out of the motor through the other. By properly sequencing the current through the phases, the motor turns. A BLDC motor is a synchronous machine so, when driven correctly, the rotational speed of the motor has a direct relationship to the rate of sequencing. In order to maintain synchronicity over a speed/torque range, the rotor position should be monitored. There are several options for feedback systems, including Hall Effect sensors and optical encoders. Figure 1 shows the connection details of a typical brushless DC motor.

Figure 1. Brushless DC Motor Wiring



Block Diagram

Figure 2 on page 5 is a complete block diagram of the control and feedback mechanisms for a brushless DC motor. Not all of these feedback mechanisms are present in every implementation; the implementation described in this application note uses the optical encoder for feedback.

Six individual Pulse Width Modulator (PWM) signals are used to drive a 3-phase inverter bridge that powers the motor. The PWM duty cycle gives a degree of control of the current in the winding. The sequencing of active PWM signals causes the motor to rotate.

In this implementation, only two of the six PWM signals will be active at any given time. The two PWM signals must be in phase. Great care must be taken to ensure that the high and low side switches on a single phase are not active simultaneously, as this causes a short circuit (known as shoot-through). The commutation sequence ensures that this does not occur. (To *commutate* is to change the current flow through the motor windings in such a way as to cause a rotating magnetic field in the motor.)

Hall Effect sensors embedded in the motor provide three digital signals that indicate the position of the rotor to within 30° (it varies from motor to motor, but is an integral dividend of 60°); this is used to determine the commutation sequence and when to cycle through the sequence. For low-resolution uses, this can also be used to compute the rotational speed of the motor. The Hall Effect sensor outputs from the motor are connected to three available GPIO pins; the interrupt generation capabilities of the GPIO pins allow for efficient commutation.

Figure 2. System Block Diagram

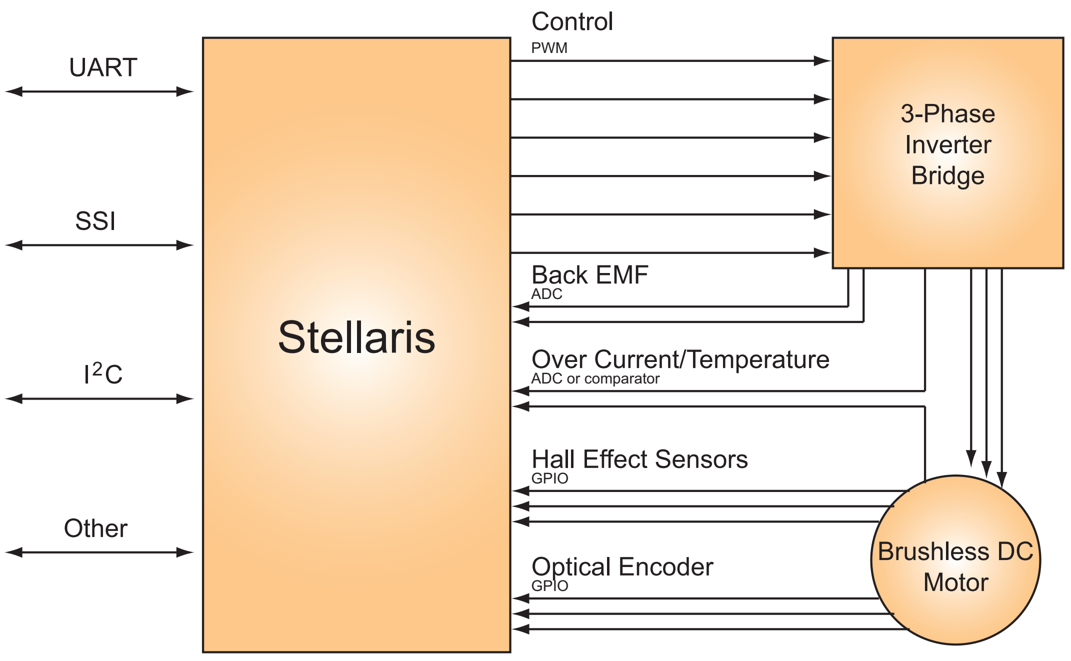


Figure 3 on page 6 shows how the commutation sequence from the Hall Effect sensors determines the PWM signals driving the 3-phase inverter bridge. This example is based on the Pittman N2341S001 motor; other BLDC motors are driven the same way, but possibly with different commutation sequences.

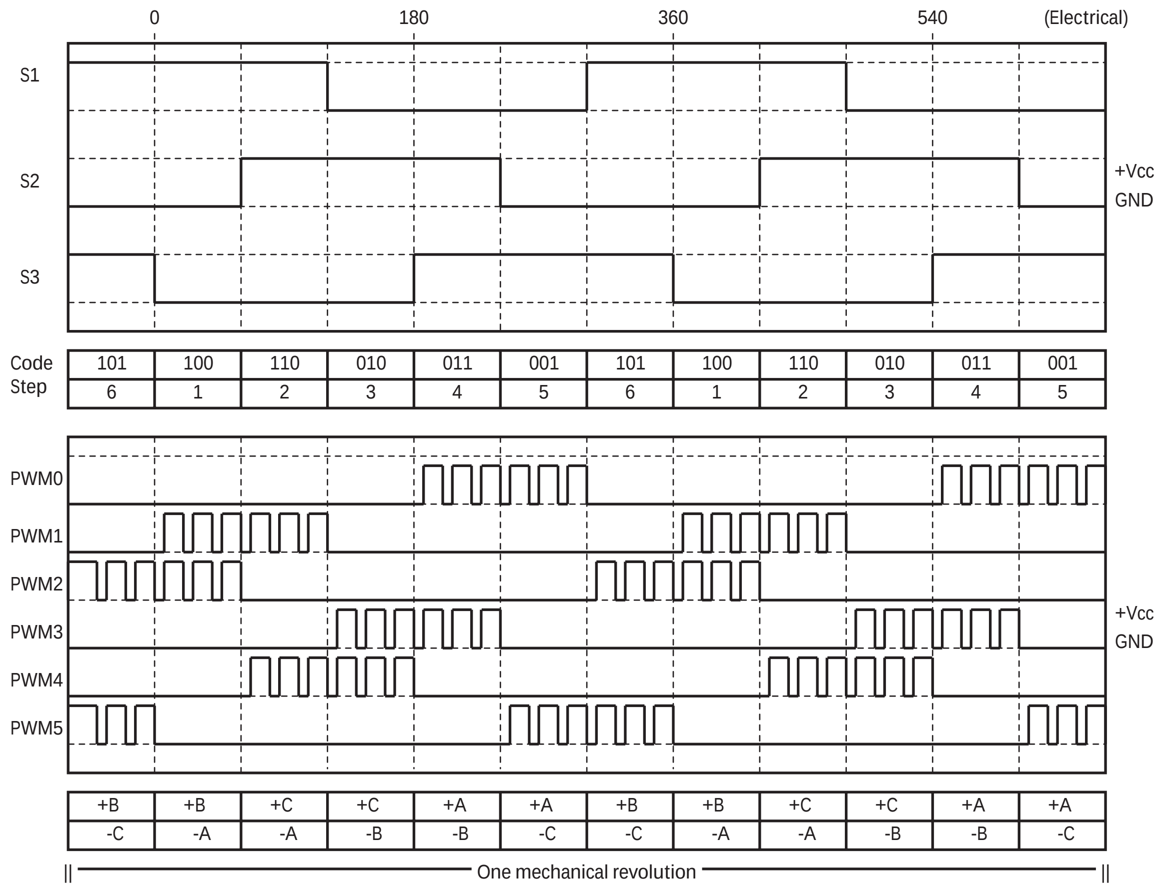
To determine the rotational speed of the motor in high-speed or high-resolution uses, an optical encoder in the motor provides high-resolution positional and velocity information. This speed information is fed into a PID (proportional, integral, and derivative gain) algorithm to adjust the PWM duty cycle as appropriate to maintain the desired speed. Tuning of the PID algorithm determines its response to changes in set speed, changes in load, and so on.

Some brushless DC motors do not have Hall Effect sensors; detection of the zero-crossing on the back EMF of the motor windings is instead used to determine when to commutate. Zero-crossing detect of the back EMF can be done in discrete components, generating a digital signal very similar to that produced by the Hall Effect sensors (though slightly different in time). The internal analog comparators can be utilized to reduce (or possibly eliminate) the number of external discretes required to do zero-crossing detection.

Over-current (via a current sensing resistor) and/or over-temperature (via a thermistor) detection should be performed, either by scaling the analog signals to appropriate levels and using an analog comparator, or by measuring them with the ADC. Over-temperature conditions may be the result of excess current in the motor or ambient conditions. Either of these conditions should result in the motor being turned off to avoid damaging it.

Several interfaces are available to provide the control input: a UART, SSI, I²C, and directly connected devices (such as switches). The interface utilized depends on the specific application in question.

Figure 3. Commutation Sequence



Software

The sections that follow describe how to use the DriverLib functions to control the BLDC motor. The accompanying zip file contains the complete source code to the example application. It should be extracted to the same directory that the DriverLib zip file was extracted in order to build it. The code will be extracted into the DriverLib/AppNotes/an01238 directory. The following discussion refers to the source, and provides abbreviated snippets of key functions within the application.

Controlling Motor Rotation with GPIOIntHandler

Basic rotational control of the motor is accomplished by responding to feedback (Hall Effect sensors, back EMF zero-crossing detect, or optical encoder) to perform commutation; the active high- and low-side transistors in the 3-phase inverter bridge are changed as appropriate when the feedback indicates the necessity. Commutation makes the motor rotate, and the commutation sequence determines the direction of rotation. The Hall Effect sensors are connected to GPIO pins, which are configured for double-edge triggered interrupt generation. Generation of an interrupt from any of those pins calls the **GPIOIntHandler** function (contained in the bldc.c file), which performs the actual motor commutation; the key elements of which are shown below.

```
void
GPIOIntHandler(void)
{
    unsigned long ulHall, ulPWM;

    //
    // Clear the GPIO pin interrupts.
    //
    GPIOPinIntClear(HALL_PORT, HALL_A | HALL_B | HALL_C);

    //
    // Perform commutation if the motor is running.
    //
    if(g_iRunning == MOTOR_RUNNING)
    {
        //
        // Get the current Hall effect sensor state.
        //
        ulHall = (GPIOPinRead(HALL_PORT, HALL_A | HALL_B | HALL_C) >>
                  HALL_SHIFT);

        //
        // Get the set of PWM signals to be driven.
        //
        ulPWM = pulHallToPhase[ulHall];

        //
        // Set the PWM signals to be driven.
        //
        PWMOutputState(PWM_BASE, ulPWM ^ (PHASE_A | PHASE_B | PHASE_C), false);
        PWMOutputState(PWM_BASE, ulPWM, true);
    }
}
```

Controlling Motor Speed with SpeedHandler

For precise control of the motor, a PID algorithm is used to vary the PWM ratio. Based on speed feedback from the optical encoder, the PWM ratio is slewed at a controlled rate to make the actual motor speed match the requested motor speed. Dampening factors in the algorithm are tuned to keep speed over- and undershoots to a minimum, and to ensure that a steady state speed is reached (that is, avoid going open-loop).

An optical encoder on the motor is used to determine the rotational speed of the motor. If present, the Stellaris microcontroller’s Quadrature Encoder Interface (QEI) block will be used to measure the motor speed; if no QEI block is present on the Stellaris device in use, a software QEI is utilized instead. For additional information about Stellaris Family devices, see the documents listed in “References” on page 17.

Thanks to the software PID controller, basic speed control of the motor is accomplished by simply varying the duty cycle of the PWM outputs based on an external request to change the motor speed (or start/stop it). A higher PWM duty-cycle increases the rate of rise in stator flux and consequently, the rotor advances faster.

To maintain synchronization between the rotor and stator fields during acceleration, the motor is slewed from the current speed to the requested speed; this is especially important when changing the direction of rotation. The **SpeedHandler** function (contained in the bldc.c file) is called in response to speed change requests from the user interface (it presently does not perform speed slewing); key elements of which are shown below.

```
static void
SpeedHandler(unsigned long ulValue)
{
    //
    // Compute the new base PWM duty cycle.
    //
    g_ulBaseDutyCycle = GetDutyCycle(ulValue);

    //
    // Save the new target speed.
    //
    g_ulTarget = ulValue;

    //
    // See if open loop operation is enabled.
    //
    if(g_bOpenLoop)
    {
        //
        // Set the PWM duty cycle based on the requested target speed.
        //
        SetPWMDutyCycle(g_ulBaseDutyCycle);
    }
}
```

Over-current and/or over-temperature monitoring may be present to prevent damaging the motor. The handling of a fault condition depends on the needs of the application; the motor could be shut off permanently, shut off until the fault condition clears, have its speed reduced to allow the fault condition to clear, and so on. In this example, fault conditions are simply ignored (they are not even detected).

Using a Timer and Event Counter for a Software QEI

The edge-triggered interrupt generation capability of the GPIO block is used to count the number of rising edges on the `ChA` output of the optical encoder; a counter is incremented on every interrupt. The **EdgeIntHandler** function (contained in the bldc.c file) handles the edge interrupts:

```
void
EdgeIntHandler(void)
{
    //
    // Clear the GPIO interrupt.
    //
    GPIOPinIntClear(GPIO_PORTC_BASE, GPIO_PIN_4);

    //
    // Increment the count of edges.
    //
    g_ulVelCount++;
}
```

Then, at a periodic rate generated by a timer, the count is used as the number of edges in a specific time period. This is converted to the motor speed in RPM and used to control the speed of the motor in closed-loop operation. The **QEIIntHandler** function (contained in the bldc.c file) handles these tasks, key elements of which are shown below.

```
void
QEIIntHandler(void)
{
    long lDelta;

    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Capture the number of edges in the previous time period, and reset the
    // count for the next time period.
    //
    lDelta = g_ulVelCount * 4;
    g_ulVelCount = 0;

    //
    // Get the speed of the motor and convert it from pulses per 1/100th of a
    / second into rpm.
    //
    g_ulSpeed = lDelta = (lDelta * 100 * 60) / ENCODER_LINES * 4;

    //
    // Update the PWM duty cycle if the motor is being driven closed-loop.
    //
    if(!g_bOpenLoop)
    {
        //
        // Compute a new duty cycle delta.
        //
        lDelta = PIDUpdate(&g_sPID, lDelta, g_ulTarget - lDelta);

        //
        // Update the PWM duty cycle.
        //
        lDelta = g_ulBaseDutyCycle + (lDelta / 10000);
        SetPWMDutyCycle(lDelta);
    }
}
```

Using the Stellaris Hardware QEI

On the LM3S601 and LM3S801 microcontrollers, a hardware QEI can be used instead of the software QEI. Time period completion interrupts from the QEI block call the same QEIIntHandler function, which performs the closed-loop speed control as in the software QEI case. This is the same function as used for software QEI, with a few conditionals added to handle the portions that are slightly different.

```
void
QEIIntHandler(void)
{
    long lDelta;

    //
    // Clear the QEI interrupt.
    //
    QEIIntClear(QEI_BASE, QEI_INTTIMER);

    //
    // Capture the encoder edge count.
    //
    lDelta = QEIGetVelocity(QEI_BASE);

    //
    // Get the speed of the motor and convert it from pulses per 1/100th of a
    // second into rpm.
    //
    g_ulSpeed = lDelta = (lDelta * 100 * 60) / (ENCODER_LINES * 4);

    //
    // The remainder of this function is the same as for the software QEI case.
    //
    ...
}
```

Providing a User Interface through the UART

A UART is used to provide the user interface to the BLDC motor application; the UART is run at 115,200 baud with an 8-N-1 character format. The UART displays a status line that looks like the following:

```
Cr s:1000 a:0996
```

The first character ("C" in this case) indicates if the motor is being run open or closed loop, with "O" indicating open loop operation and "C" indicating closed loop operation. The second character ("r" in this case) indicates if the motor is running or stopped, with "s" indicating stopped and "r" indicating running. The number after the "s:" is the requested motor speed; in this example, the motor has been asked to run at 1000 rpm. The number after the "a:" is the measured motor speed; in this example, the motor is running at 996 rpm. The actual speed display is updated approximately every 1/3 of a second.

When first started, the set speed is 0 rpm, the motor is stopped, and it is configured for open loop operation. The default PID factors are tuned for good response to load changes (such as pressing the motor spindle with your finger).

Table 1 describes the commands that can be sent to the application.

Table 1. BLDC Motor Application Commands

Character	Description
r	Starts/stops the motor. If the motor is running, this command stops it. If it is stopped, this command causes it to start running in the opposite direction than it was previously running. Therefore, back-to-back "r" commands cause the motor to change its direction of rotation.
l	Opens or closes the feedback loop. When in open-loop operation, a PWM duty cycle is chosen based on the requested speed. When in closed-loop operation, the base PWM duty cycle is chosen the same way, but is adjusted by the PID algorithm.
p	Sets a new proportional gain factor for the PID algorithm.
i	Sets a new integral gain factor for the PID algorithm.
d	Sets a new derivative gain factor for the PID algorithm.
z	Displays the current PID gain factors.
D	When in closed loop operation, this displays a long stream of the measured motor speed, one data sample every 1/100th of a second. This can be used to evaluate the performance of the gain factors in the PID algorithm; speed oscillations are easier to see in the numbers than to observe by watching the motor. Additionally, with some post-massaging of the text, this data can be imported into something like Excel and graphed, making the motor performance very easy to see.

Memory Size

The application requires approximately 5.5 KB of Flash and 0.5 KB of SRAM and easily runs at a 6 MHz processor clock. Given the 16 KB of Flash, 4 KB of SRAM, and 25 MHz processor clock on the LM3S310 (the smallest Stellaris microcontroller with the required peripheral set for this application), there is plenty of headroom for additional functionality or more complicated motor control mechanisms.

Schematics

Figure 4 starting on page 13 through Figure 7 on page 16 provide example schematics for connecting a brushless DC motor with Hall Effect sensors and an optical encoder to an LM3S601 microcontroller. The six PWM signals are used to drive the three half-H bridges, one per motor winding. GPIO pins are used to connect the Hall Effect sensors, and the optical encoder is connected to the QEI module. An RS-232 driver is connected to the first UART for use by the user interface.

Figure 4. Brushless DC Motor Control: MCU

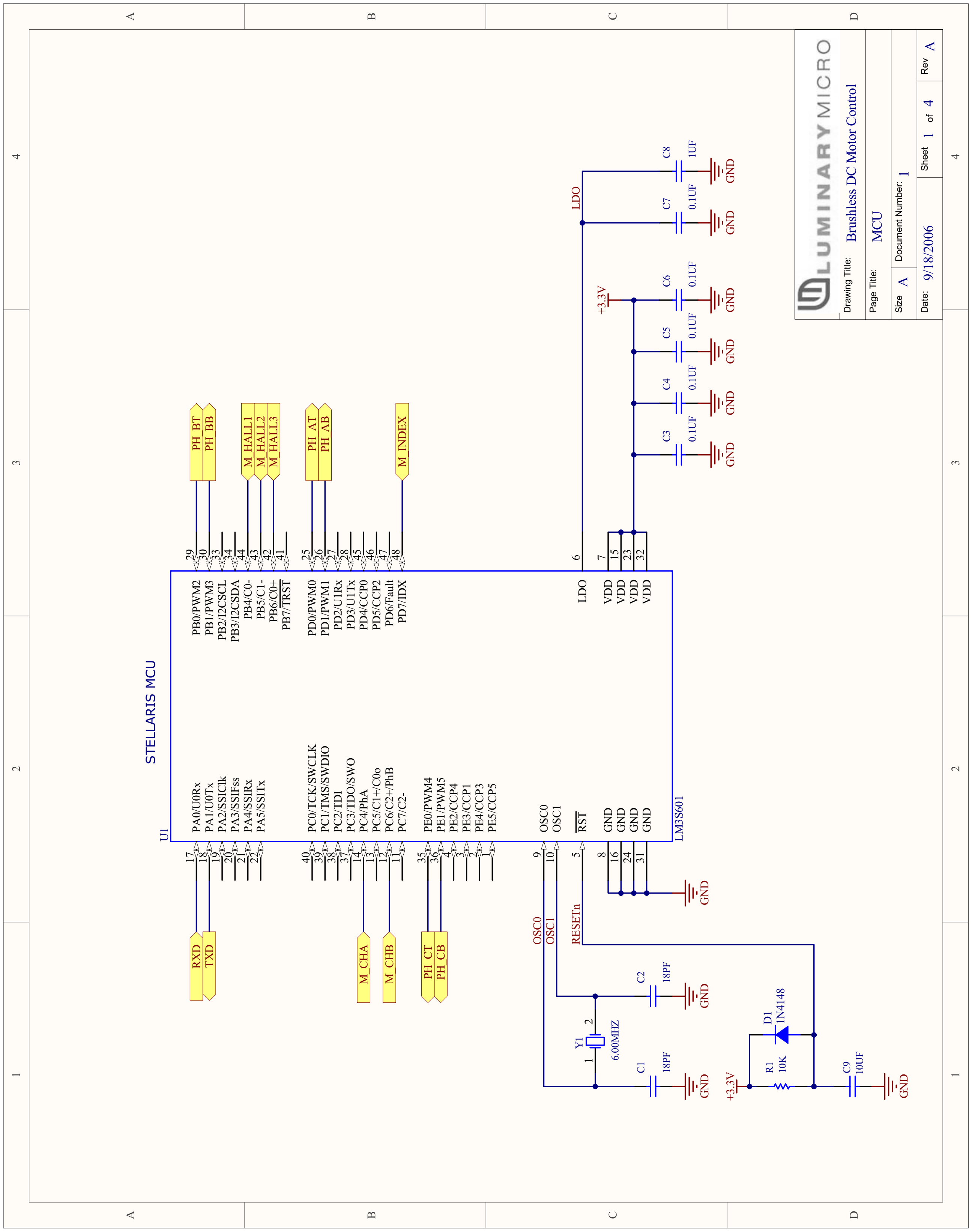
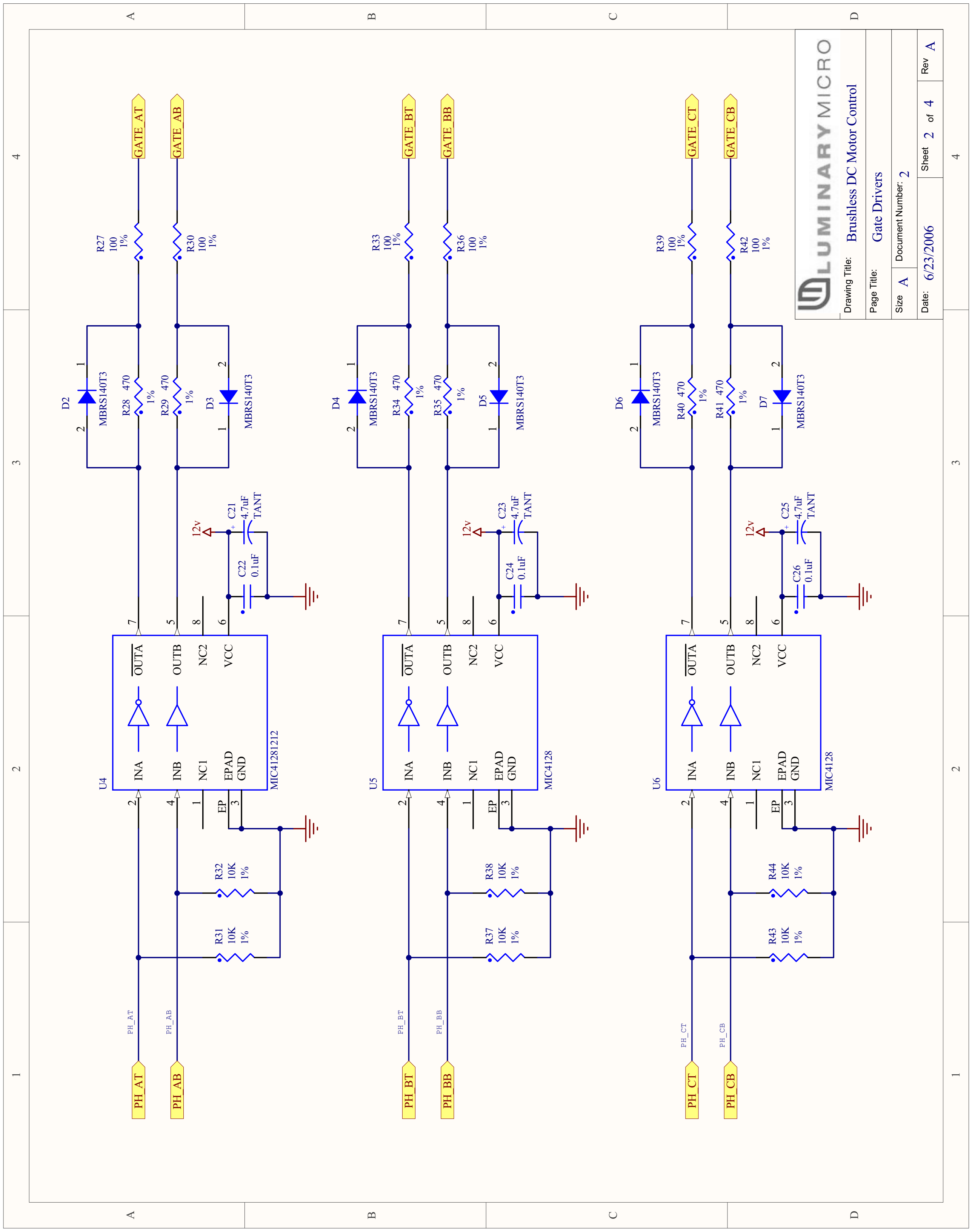


Figure 5. Brushless DC Motor Control: Gate Drivers



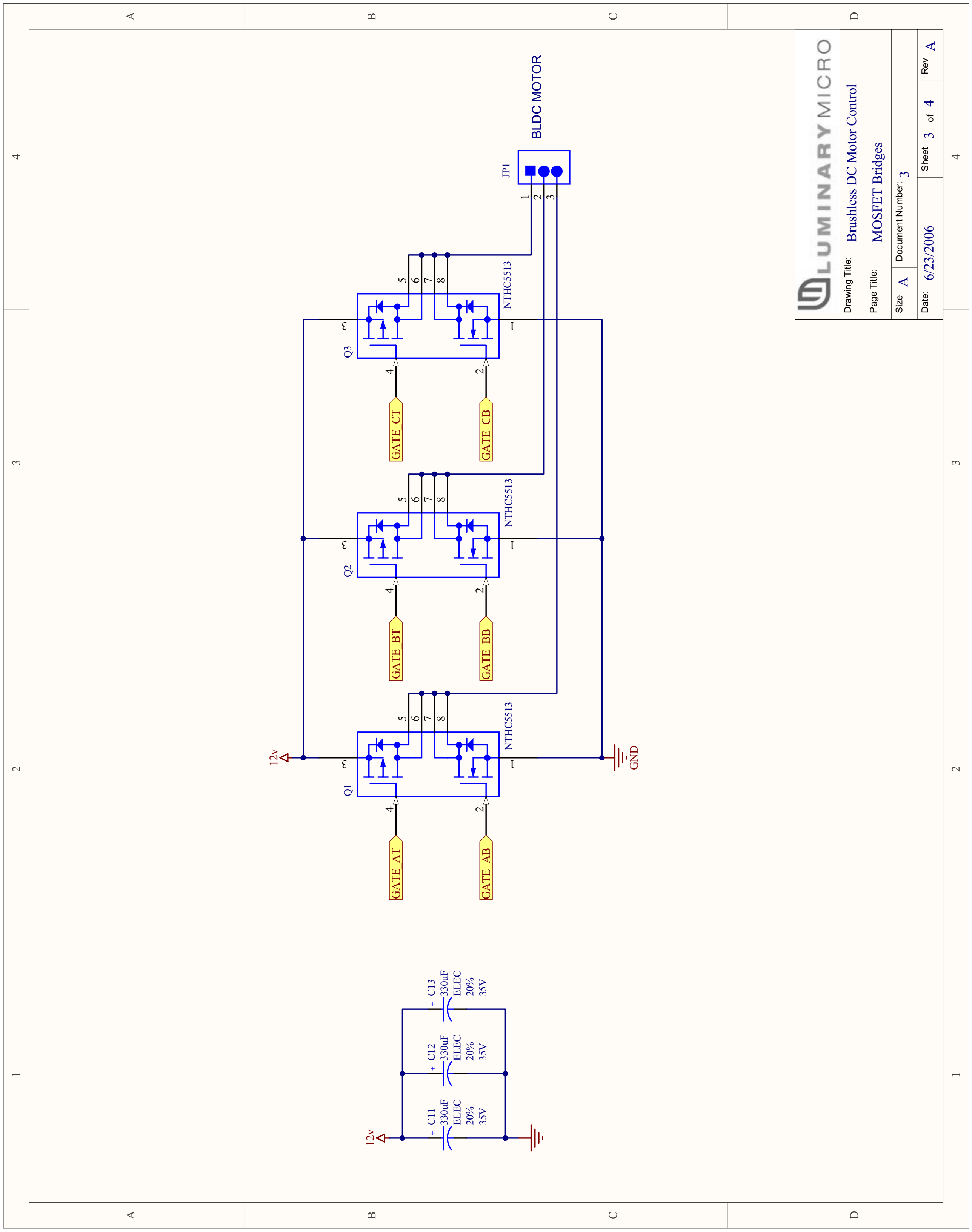
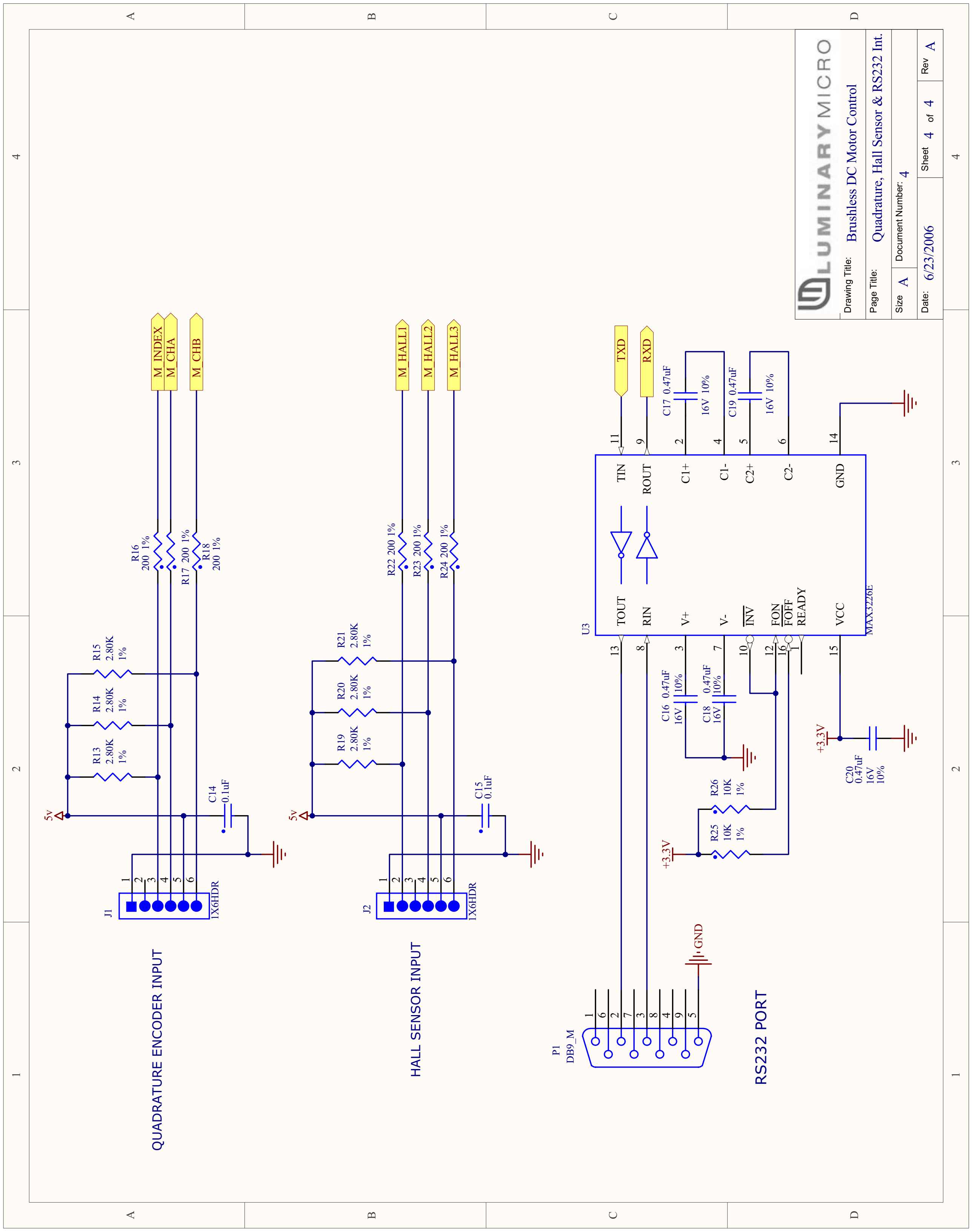


Figure 6. Brushless DC Motor Control: MOSFET Bridges

Figure 7. Brushless DC Motor Control: Quadrature, Hall Sensor, and RS232 Interface



This design does not provide details of the DC power supply section required to power the processor, sensors, and bridges/motor. Great care must be taken in the power supply section to provide enough current to drive the motor, but limit the current so that the bridge and/or motor are not damaged. A typical design would provide current sensing to detect over-current situations and adjust the motor's operation accordingly. The "Fault" input pin on the Stellaris microcontroller can be utilized to provide a rapid shutdown of the PWM drives in response to a hardware detected error condition.

Although the LM3S601 microcontroller is shown in the schematic, the LM3S801 microcontroller could be used as well while still utilizing the QEI module. Additionally, the LM3S310, LM3S610, LM3S611, LM3S615, LM3S811, or LM3S815 microcontrollers could also be used (due to their six PWM channels) with the software QEI solution. If the LM3S610, LM3S611, LM3S615, LM3S811, or LM3S815 microcontrollers are used, the on-chip ADC can be utilized to monitor the bridge current for detection of over-current conditions.

Conclusion

The Stellaris family of microcontrollers contains a set of peripherals that allow a brushless DC motor to be easily controlled. Both open- and closed-loop operation is possible, with plenty of overhead remaining for other system-level operations (such as communicating with other devices within the target application or performing more sophisticated closed-loop control).

References

The following are available for download at www.luminarymicro.com:

- Stellaris microcontroller data sheet, Publication Number DS-LM3S*nnn* (where *nnn* is the part number for that specific Stellaris family device)
- *Stellaris™ Family Driver Library User's Manual*, Publication Number SW02032
- Stellaris Family Driver Library
- *Stellaris™ Family Product Selector Guide*, Publication Number PSG-LM3SFAM-02

Company Information

Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3 based microcontrollers for use in embedded applications within the industrial, commercial, and consumer markets. Luminary Micro is ARM's lead partner in the implementation of the Cortex-M3 core. Please contact us if you are interested in obtaining further information about our company or our products.

Luminary Micro, Inc.
2499 South Capital of Texas Hwy, Suite A-100
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com
sales@luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext. 3