Why the motorShield\_v3 was made the way it is

**INTRODUCTION**

the motor shield was designed for the very specific purpose of a project I developed in the field of haptics where people were in the need of controlling DC motors with encoders. At the time, Arduino's IDE had no support yet for interrupts and therefore the code I generated was far too complex to be published. Now, 3 years and 2 laptops later, I have a hard time to find the results to that project.

However, here my 50cents.

**TECHNICAL DESCRIPTION**

The issue with the Arduino board is that we have only 2 pins where to attach external hardware interrupts. These pins are 2 and 3. When using encoders, a traditional approach consists in using one interrupt pin per encoder, since you need to detect two pins generating pulses for each encoder. They are usually labeled as A and B. The arrival of a pulse to one of those pins indicates the rotation's direction, while the amount of pulses per second shows the speed.

In theory, for encoders with small amounts of steps it should be possible to read the mentioned data with just digitalRead commands. But this would make our program dependent in having a very optimized way to handle the whole code. Therefore an interrupt driven program is the way to read encoders.

The IC used on the motor shield can be used to drive one stepper motor, or two DC motors controlling speed and direction, or four DC motors controlling with fixed direction. The encoder add one has been designed to allow one stepper, or up to two DC motors to include encoders if we measure direction and speed, or four DC motors if we measure only speed. This means a maximum of 4 encoder-generated signals to attack only two interrupt pins.

The trick here is to multiplex the interrupts into one of the pins via OR gates. Therefore the 74XX32 IC that is part of the partlist. Whenever any of the pins in all the encoders generates an event, it will go into the OR gate and the interrupt function (as declared with attachInterrupt) will be called. Once in the Interrupt Handling Routine, it is possible to read the different pins and determine which was the one that triggered the event. That is why the encoder pins are mapped both to the OR gate but also to other pins.

On the schematic you will see that two of the encoder pins are labeled as EC1 and EC2, the other two are connected to the on board buttons. I made it this way because in the project I was working we realized that having extra buttons on the board would be needed to select certain operations. You can anyway take cables from the pushbutton connectors to your encoders instead.

Finally, I decided to add the pin labeled E3 to give access to interrupt 1, some people may be interested in using that for some other interrupt driven code.

**WHAT ABOUT THE Exy PINS?**

Those pins allow you switching between a stepper motor or DC ones. When left open, the logic gates will take care of switching speed and direction of the DC motors. When connected (E1 with E12, and E2 with E22), the driver IC will be used as just a series of power transistors to drive the stepper.

**HARDWARE**

Let me tell you about the chips:

* L293: this is the motor driver, it controls the motors, the direction of rotation, and the speed with just a couple of pins
* the 74XXX1 is a NAN gate, this will allow controlling the motors with only a couple of pins instead of having to attack the driver chip with 4 pins per motor, you can do it with two, it is there to make your life easier
* the 74XX32 is an OR gate, it is used to multiplex the arrival of interrupts (as I explained in the standard email)

**SOFTWARE**

As mentioned, I have no code example for this using the current stat of the art of Arduino's IDE. But I can sketch one for you to use.

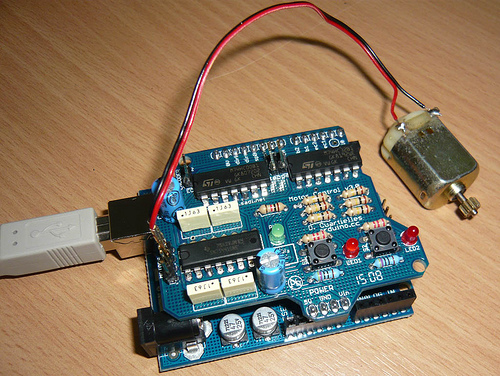
controlling speed and direction of a small DC motor

**Introduction**

This example shows how to control a small DC motor using the motor shield v3 for Arduino compatible boards.

**Scenario**

The scenario of use is as shown on this picture:

[](http://www.flickr.com/photos/dcuartielles/3343322313/)

*(c) 2009 Picture courtesy of Rui Costa*

**Code**

The following sketch for Arduino shows how to control the speed and direction of the motor connected as shown in the previous picture.

int dirA = 12;

int dirB = 13; // not used in this example

int speedA = 10;

int speedB = 11; // not used in this example

void setup()

{

pinMode (dirA, OUTPUT);

pinMode (dirB, OUTPUT);

pinMode (speedA, OUTPUT);

pinMode (speedB, OUTPUT);

}

void loop()

{

// move the motor A to one direction increasing speed

digitalWrite (dirA, HIGH);

for (int j = 0; j < 255; j += 10)

{

analogWrite (speedA, j);

delay (100);

}

delay(1000); // keep the motor rolling for one second

// move the motor A to one direction decreasing speed

digitalWrite (dirA, HIGH);

for (int j = 255; j >= 0; j -= 10)

{

analogWrite (speedA, j);

delay (100);

}

// stop the motor

digitalWrite(speedA, LOW);

delay(1000); // keep the motor stopped for one second

}

## First Sketch

In the getting started guide ([Windows](http://arduino.cc/en/Guide/Windows), [Mac OS X](http://arduino.cc/en/Guide/MacOSX), [Linux](http://www.arduino.cc/playground/Learning/Linux)), you uploaded a sketch that blinks an LED. In this tutorial, you'll learn how each part of that sketch works.

### Sketch

A sketch is the name that Arduino uses for a program. It's the unit of code that is uploaded to and run on an Arduino board.

### Comments

The first few lines of the [Blink](http://arduino.cc/en/Tutorial/Blink) sketch are a comment:

/\*

\* Blink

\*

\* The basic Arduino example. Turns on an LED on for one second,

\* then off for one second, and so on... We use pin 13 because,

\* depending on your Arduino board, it has either a built-in LED

\* or a built-in resistor so that you need only an LED.

\*

\* http://www.arduino.cc/en/Tutorial/Blink

\*/

Everything between the /\* and \*/ is ignored by the Arduino when it runs the sketch (the \* at the start of each line is only there to make the comment look pretty, and isn't required). It's there for people reading the code: to explain what the program does, how it works, or why it's written the way it is. It's a good practice to comment your sketches, and to keep the comments up-to-date when you modify the code. This helps other people to learn from or modify your code.

There's another style for short, single-line comments. These start with // and continue to the end of the line. For example, in the line:

int ledPin = 13; // LED connected to digital pin 13

the message "LED connected to digital pin 13" is a comment.

### Variables

A variable is a place for storing a piece of data. It has a name, a type, and a value. For example, the line from the Blink sketch above declares a variable with the name ledPin, the type int, and an initial value of 13. It's being used to indicate which Arduino pin the LED is connected to. Every time the name ledPin appears in the code, its value will be retrieved. In this case, the person writing the program could have chosen not to bother creating the ledPin variable and instead have simply written 13 everywhere they needed to specify a pin number. The advantage of using a variable is that it's easier to move the LED to a different pin: you only need to edit the one line that assigns the initial value to the variable.

Often, however, the value of a variable will change while the sketch runs. For example, you could store the value read from an input into a variable. There's more information in the [Variables tutorial](http://arduino.cc/en/Tutorial/Variables).

### Functions

A function (otherwise known as a procedure or sub-routine) is a named piece of code that can be used from elsewhere in a sketch. For example, here's the definition of the setup() function from the Blink example:

void setup()

{

pinMode(ledPin, OUTPUT); // sets the digital pin as output

}

The first line provides information about the function, like its name, "setup". The text before and after the name specify its return type and parameters: these will be explained later. The code between the { and } is called the body of the function: what the function does.

You can call a function that's already been defined (either in your sketch or as part of the [Arduino language](http://arduino.cc/en/Reference/HomePage)). For example, the line pinMode(ledPin, OUTPUT); calls the pinMode() function, passing it two parameters: ledPin and OUTPUT. These parameters are used by the pinMode() function to decide which pin and mode to set.

### pinMode(), digitalWrite(), and delay()

The pinMode() function configures a pin as either an input or an output. To use it, you pass it the number of the pin to configure and the constant INPUT or OUTPUT. When configured as an input, a pin can detect the state of a sensor like a pushbutton; this is discussed in a [later tutorial?](http://arduino.cc/en/Tutorial/DigitalInput?action=edit). As an output, it can drive an actuator like an LED.

The digitalWrite() functions outputs a value on a pin. For example, the line:

digitalWrite(ledPin, HIGH);

set the ledPin (pin 13) to HIGH, or 5 volts. Writing a LOW to pin connects it to ground, or 0 volts.

The delay() causes the Arduino to wait for the specified number of milliseconds before continuing on to the next line. There are 1000 milliseconds in a second, so the line:

delay(1000);

creates a delay of one second.

### setup() and loop()

There are two special functions that are a part of every Arduino sketch: setup() and loop(). The setup() is called once, when the sketch starts. It's a good place to do setup tasks like setting pin modes or initializing libraries. The loop() function is called over and over and is heart of most sketches. You need to include both functions in your sketch, even if you don't need them for anything.

### Exercises

**1. Change the code so that the LED is on for 100 milliseconds and off for 1000.**

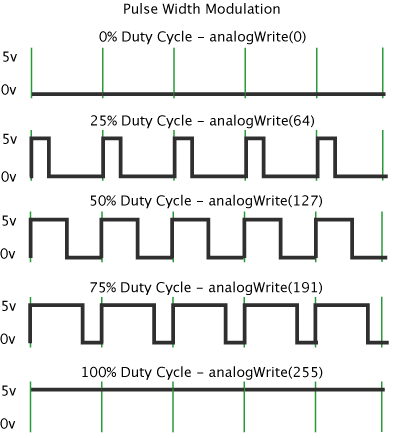
**2. Change the code so that the LED turns on when the sketch starts and stays on.**

**PWM**

The Fading example demonstrates the use of analog output (PWM) to fade an LED. It is available in the File->Sketchbook->Examples->Analog menu of the Arduino software.

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (5 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 5v controlling the brightness of the LED.

In the graphic below, the green lines represent a regular time period. This duration or period is the inverse of the PWM frequency. In other words, with Arduino's PWM frequency at about 500Hz, the green lines would measure 2 milliseconds each. A call to [analogWrite](http://arduino.cc/en/Reference/AnalogWrite)() is on a scale of 0 - 255, such that analogWrite(255) requests a 100% duty cycle (always on), and analogWrite(127) is a 50% duty cycle (on half the time) for example.



Once you get this example running, grab your arduino and shake it back and forth. What you are doing here is essentially mapping time across the space. To our eyes, the movement blurs each LED blink into a line. As the LED fades in and out, those little lines will grow and shrink in length. Now you are seeing the pulse width.

## Memory

There are three pools of memory in the microcontroller used on Arduino boards (ATmega168):

* Flash memory (program space), is where the Arduino sketch is stored.
* SRAM (static random access memory) is where the sketch creates and manipulates variables when it runs.
* EEPROM is memory space that programmers can use to store long-term information.

Flash memory and EEPROM memory are non-volatile (the information persists after the power is turned off). SRAM is volatile and will be lost when the power is cycled.

The ATmega168 chip has the following amounts of memory:

Flash 16k bytes (of which 2k is used for the bootloader)

SRAM 1024 bytes

EEPROM 512 bytes

Notice that there's not much SRAM available. It's easy to use it all up by having lots of strings in your program. For example, a declaration like:

char message[] = "I support the Cape Wind project.";

puts 32 bytes into SRAM (each character takes a byte). This might not seem like a lot, but it doesn't take long to get to 1024, especially if you have a large amount of text to send to a display, or a large lookup table, for example.

If you run out of SRAM, your program may fail in unexpected ways; it will appear to upload successfully, but not run, or run strangely. To check if this is happening, you can try commenting out or shortening the strings or other data structures in your sketch (without changing the code). If it then runs successfully, you're probably running out of SRAM. There are a few things you can do to address this problem:

* If your sketch talks to a program running on a (desktop/laptop) computer, you can try shifting data or calculations to the computer, reducing the load on the Arduino.
* If you have lookup tables or other large arrays, use the smallest data type necessary to store the values you need; for example, an [int](http://arduino.cc/en/Reference/Int) takes up two bytes, while a [byte](http://arduino.cc/en/Reference/Byte) uses only one (but can store a smaller range of values).
* If you don't need to modify the strings or data while your sketch is running, you can store them in flash (program) memory instead of SRAM; to do this, use the [PROGMEM](http://www.arduino.cc/en/Reference/PROGMEM) keyword.

To use the EEPROM, see the [EEPROM library](http://www.arduino.cc/en/Reference/EEPROM).