Understanding the Adafruit Motor Shield Library

I've recently been working on a small robotics project using the Arduino platform with the Adafruit motor shield. I'm an experienced applications programmer but a complete novice at electronics and embedded programming, so I find it challenging to understand how the hardware design and the code relate to each other. I've been going through the motor shield schematic and the code library to try to understand them both in detail, and I thought it might be helpful to other beginning Arduinists if I were to write up what I've figured out so far. At least for now I will focus on DC motor control because that's what I've been working on and have learned the most about.

Basic DC Motor Control

Let's set aside speed control for a moment and first walk through how the motors are turned on and off.

The Motor Drivers and Shift Register

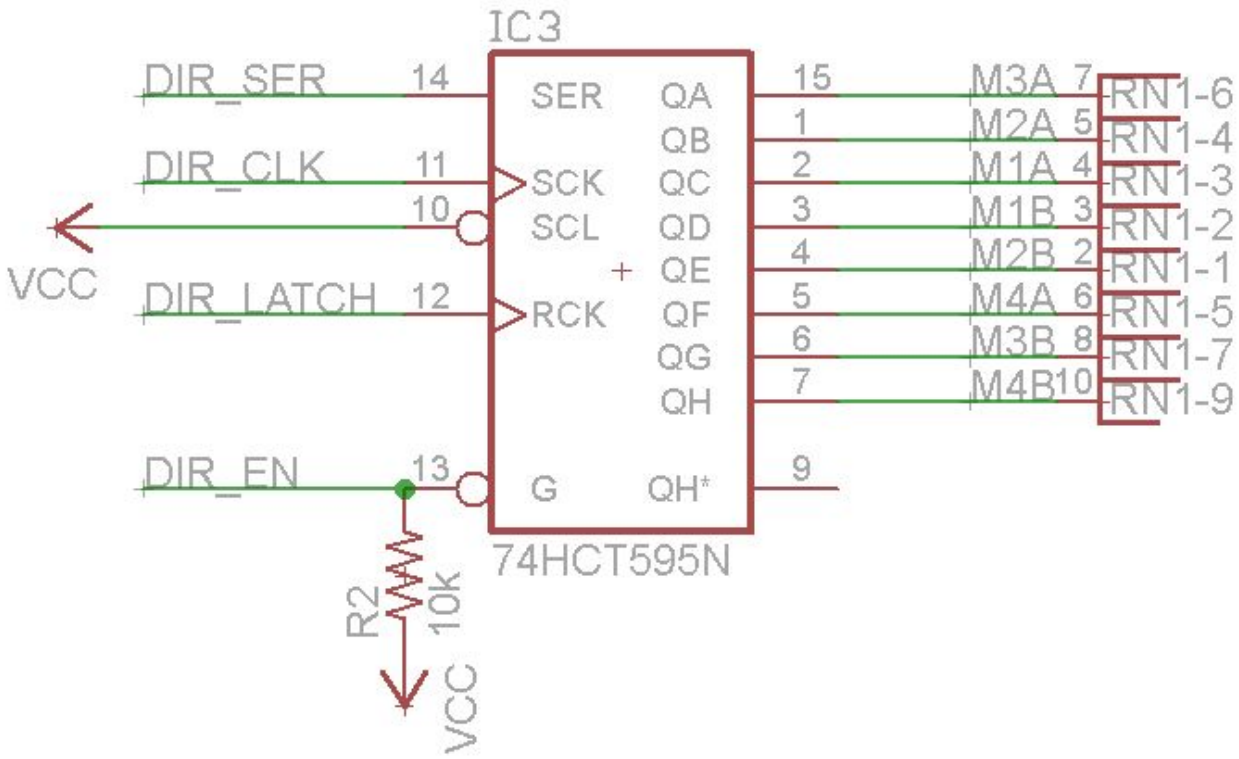
The motor shield can control up to four DC motors using two L293D dual H-bridge motor driver ICs. You can read the datasheet for details, but the main thing to know is that each IC has four digital inputs and can control two motors bidirectionally. The inputs are in pairs, and each pair controls a motor. For example, if pin 2 is set high and pin 7 is set low, motor 1 is turned on. If the inputs are reversed, the motor is again on but its direction of rotation is reversed, and if both are set low then the motor is off. There are other possible configurations but this is the one supported by the motor shield.

So obviously the thing to do is wire eight pins from the Arduino to the eight inputs on the two ICs, right? That's what I imagined at first, but it's WRONG!! Pins are a scarce resource, and the motor shield design makes an extra effort to conserve them. The inputs to the motor drivers are actually wired to the outputs of a 74HCT595N 8-bit shift register. This IC lets you feed in bits serially, but it then presents them as parallel outputs. It's controlled by four digital inputs, so effectively it lets you control eight inputs to the motor drivers (thus four motors) using four pins of the Arduino.

At this point we can start to look at code. Here are some constants defined in AFMotor.h:

```
#define MOTOR1_A 2
#define MOTOR1_B 3
#define MOTOR2_A 1
#define MOTOR2_B 4
#define MOTOR4_A 0
#define MOTOR4_B 6
#define MOTOR3_A 5
```

`#define MOTOR3_B 7`Clearly, MOTOR1_A and MOTOR1_B have to do with controlling motor 1 bidirectionally, MOTOR2_A and MOTOR2_B are related to motor 2, etc. But what do the values mean? It turns out these are the bit numbers in the output of the shift register, as you can see in this detail of the schematic:



The pins labeled QA through QH are the parallel output pins, and you can think of them as bits 0 through 7. (The numbers immediately to the right of the IC are pin numbers rather than bit numbers, although they mostly coincide. QA is bit zero. The datasheet calls them Q0 through Q7 so I'm not sure why the schematic uses a different convention).

As you can see, Motor 1 is controlled by bits 2 and 3, Motor 2 by bits 1 and 4, etc. Motors 3 and 4 are actually reversed between the code and the schematic, but I'm pretty sure that's just a bug.

The pins shown on the left are the four input pins plus the supply voltage. It takes some study of the datasheet to understand the inputs, so here's a brief description of each:

- DIR_EN -- the enable input, active low; it has a pull-up resistor to disable the outputs until an AFMotorController is created in the code, as we'll see.
- DIR_SER -- the data input.
- DIR_CLK -- A rising edge on this input causes the data bit to be shifted in and all bits shifted internally.
- DIR_LATCH -- A rising edge causes the internally-stored data bits to be presented to

the parallel outputs. This allows an entire byte to be shifted in without causing transient changes in the parallel outputs.

Almost ready to look at more code! But first a brief digression.

Ports

Arduino tutorials like Ladyada's teach you how to read and write pins one at a time using digitalRead and digitalWrite, but it turns out there's also a way to read and write them in groups, called "ports". You should definitely read this explanation, but here are the salient points:

- PORTD maps to digital pins 0 to 7, but you shouldn't use bits 0 and 1.
- PORTB maps to digital pins 8 to 13; the two high-order bits are not used.
- There are three C macros corresponding to each port:
  - DDRB/DDRD -- these are bitmasks that control whether each pin in the port is an input or output. 1 means output. Assigning a value to one of these is like calling pinMode once for each pin in the port.
  - PORTB/PORTD -- if you read one of these, it's like doing a digitalRead on each pin in the port. If you assign to one of them, it's like doing a digitalWrite on each pin.
  - PINB/PIND -- like PORTB/PORTD but read-only.

When I say setting DDRB is like calling pinMode a bunch of times, I mean it has the same end result. However, it sets the mode on all the pins all at once so it's inherently faster than multiple calls to pinMode. The same remark applies to PORTB/PORTD as opposed to digitalRead and digitalWrite.

Defining the Shift Register Inputs

Okay, finally we can understand this fragment from AFMotor.h:

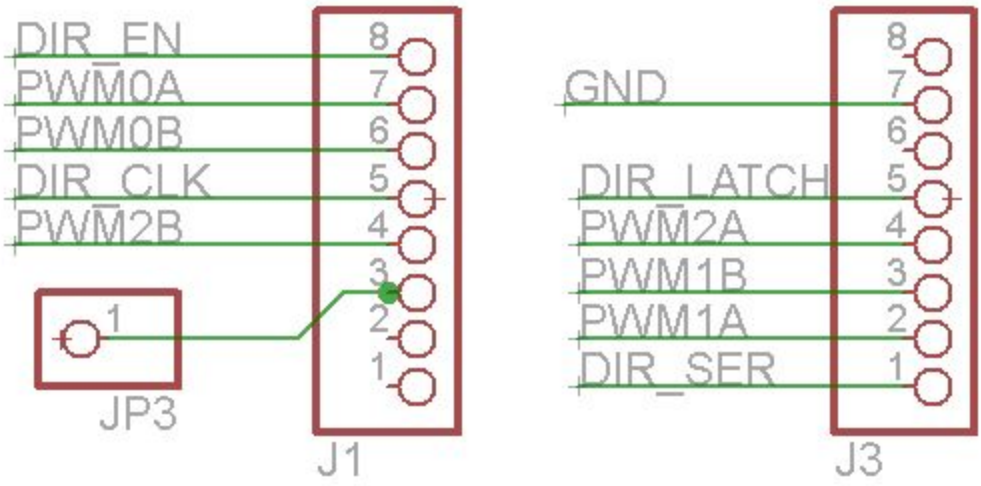
```
 #define LATCH 4
 #define LATCH_DDR DDRB
 #define LATCH_PORT PORTB

 #define CLK_PORT PORTD
 #define CLK_DDR DDRD
 #define CLK 4

 #define ENABLE_PORT PORTD
 #define ENABLE_DDR DDRD
 #define ENABLE 7

 #define SER 0
 #define SER_DDR DDRB
#define SER_PORT PORTB
```

Clearly these four groups of #defines correspond one-to-one with the inputs to the shift register, and specifically they define the relationship between Arduino pins and shift register

inputs:

Shift register input	Port	Bit # of port	Arduino digital pin #
DIR_LATCH	B	4	12
DIR_CLK	D	4	4
DIR_EN	D	7	7
DIR_SER	B	0	8

We'd better check this against the schematic and make sure we're understanding it correctly:



J1 corresponds to digital pins 0-7, and J3 to pins 8-14 (which you can verify on the Arduino schematic), so it looks like the schematic and the code line up.

AFMotorController

The mechanics of pushing values out to the shift register are encapsulated in a C++ class named AFMotorController. There is an instance of this class declared in AFMotor.cpp; this instance is used by the classes AF_DCMotor and AF_Stepper.

AFMotorController declares two member functions: enable() and latch_tx(). The latch_tx() function is responsible for updating the outputs of the shift register with the bits of the global variable latch_state, which is declared as:

```
static uint8_t latch_state;
```

Here's the body of latch_tx() with pseudocode comments added by me:

```
 void AFMotorController::latch_tx(void) {
   uint8_t i;

   LATCH_PORT &= ~_BV(LATCH);   // - Set DIR_LATCH low
   SER_PORT &= ~_BV(SER);          // - Set DIR_SER low
   for (i=0; i<8; i++) {       // - For each bit of latch_state:
     CLK_PORT &= ~_BV(CLK);      //    - Set DIR_CLK low

     if (latch_state & _BV(7-i))  //    - If the current bit of
                                  //      latch_state is set:
       SER_PORT |= _BV(SER);      //       - Set DIR_SER high
     else                         //       else:
       SER_PORT &= ~_BV(SER);     //       - Set DIR_SER low

     CLK_PORT |= _BV(CLK);        //    - Set DIR_CLK high; the
                                  //      rising edge shifts DIR_SER
                                  //      into the shift register
   }
   LATCH_PORT |= _BV(LATCH);      // - Set DIR_LATCH high; the
                                  //   rising edge sends the stored
                                  //   bits to the parallel outputs
}
```

This function looks rather cryptic at first glance, but the key is that _BV(i) is a macro which evaluates to a byte having only the *i*th bit set. It's defined in avr/str_defs.h as: `#define _BV(bit) (1 << (bit))` Knowing that, you can easily work out how the following two statements set a single bit of SER_PORT to high and low respectively. The same idiom is used everywhere in this code so it's worthwhile to think through it if you're not used to doing bit manipulations.

```
 SER_PORT |= _BV(SER);
SER_PORT &= ~_BV(SER);
```

The enable() function initializes the ports, then clears and enables the shift register outputs. Until the outputs are enabled, they are in the high impedance state, i.e. effectively turned off. This function is simple, so here it is verbatim (including original comments):

```
 void AFMotorController::enable(void) {
   // setup the latch
   LATCH_DDR |= _BV(LATCH);
   ENABLE_DDR |= _BV(ENABLE);
   CLK_DDR |= _BV(CLK);
   SER_DDR |= _BV(SER);
   latch_state = 0;

   latch_tx();  // "reset"

   ENABLE_PORT &= ~_BV(ENABLE); // enable the chip outputs!
}
```

AF_DCMotor

As described in the motor shield library documentation, you control a motor by instantiating an AF_DCMotor object with the appropriate motor number (1 through 4), then use the run() function to turn the motor on and off. AF_DCMotor also has a setSpeed() function, but we'll return to that in the next section when we look at speed control.

Here's the run() function with a few comments from me. This function breaks naturally into two sections; one in which we figure out which shift register outputs need to be set, and one in which we set them.

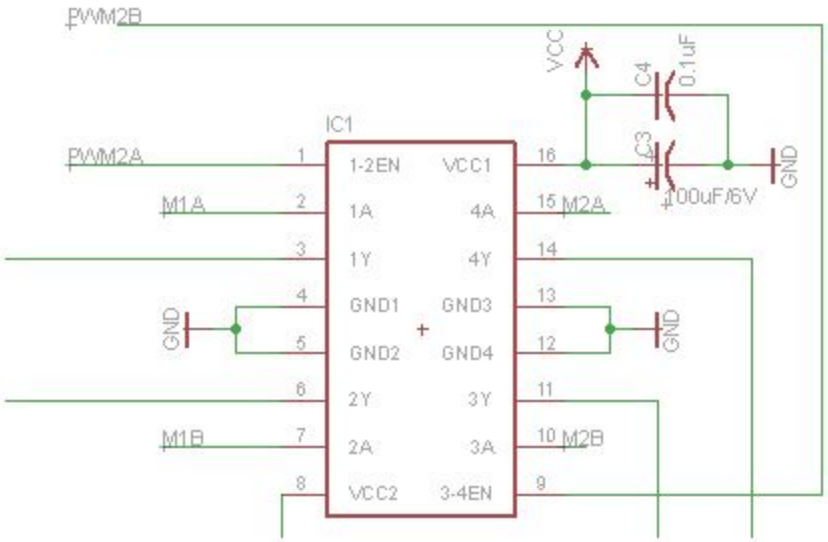
```
 void AF_DCMotor::run(uint8_t cmd) {
   uint8_t a, b;

   /* Section 1: choose two shift register outputs based on which
    * motor this instance is associated with.   motornum is the
    * motor number that was passed to this instance's constructor.
    */
   switch (motornum) {
   case 1:
     a = MOTOR1_A; b = MOTOR1_B; break;
   case 2:
     a = MOTOR2_A; b = MOTOR2_B; break;
   case 3:
     a = MOTOR3_A; b = MOTOR3_B; break;
   case 4:
     a = MOTOR4_A; b = MOTOR4_B; break;
   default:
     return;
   }

   /* Section 2: set the selected shift register outputs to high/low,
    * low/high, or low/low depending on the command.  This is done
    * by updating the appropriate bits of latch_state and then
    * calling tx_latch() to send latch_state to the chip.
    */
   switch (cmd) {
   case FORWARD:               // high/low
     latch_state |= _BV(a);
     latch_state &= ~_BV(b);
     MC.latch_tx();
     break;
   case BACKWARD:              // low/high
     latch_state &= ~_BV(a);
     latch_state |= _BV(b);
     MC.latch_tx();
     break;
   case RELEASE:               // low/low
     latch_state &= ~_BV(a);
     latch_state &= ~_BV(b);
     MC.latch_tx();
     break;
   }
}
```

Speed Control

The speed of the DC motors is controlled using pulse-width-modulation (PWM). The idea of PWM is to control the power to a motor using a high-frequency square wave. When the

square wave signal is on the motor is powered, and when the signal is low the power is turned off. The speed of the motor is controlled by the fraction of time the controlling signal is on. The Arduino can generate square waves for PWM on pins 3, 5, 6, 9, 10, 11.

The wiring of the PWM pins for speed control is straightforward. On the motor driver ICs, in addition to the two control inputs for each motor, there is an enable input. If it is set low then the motor is not powered regardless of the state of the other two pins. Four of the PWM pins are wired directly to the four enable inputs on the two motor drivers. This detail from the schematic shows two PWM inputs to the driver for motors 1 and 2.



While the wiring is easy to understand, the code requires some explanation. It directly accesses internal registers of the ATmega processor, and the only detailed documentation I could find was in the datasheet. So let's dive in.

The ATmega processor has three timer/counter modules that can be used to generate a PWM signal. They are numbered 0, 1, and 2, and I'll refer to them as TC0, TC1, TC2. Conveniently the motor shield schematic uses the same numbering, so in the above figure you can see that TC2 is being used to control motors 1 and 2. Basically, the way the TC modules work is that a clock increments a counter on each clock cycle. Each TC has two associated comparison registers and two associated output pins; a pin can be set low or high depending on a comparison between the counter and the corresponding register.

The modules can be configured in various operating modes, but in the "Fast PWM" mode that is used by the library, the output pin is set high each time the counter sets back to zero and set low when the counter reaches the comparison value, which therefore acts as the speed setting. TC0 and TC2 each have an 8-bit counter, which is why the speed range for the motors is 0-255.

Each TC module is controlled by two registers. TC2 is controlled by registers TCCR2A and TCCR2B, which have the following bit layouts:

Bit	7	6	5	4	3	2	1	0	
(0xB0)	COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20	TCCR2A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
(0xB1)	FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20	TCCR2B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The comparison registers are identified as OCR2A and OCR2B. The avr headers define assignable macros with names identical to the register names as given in the datasheet. With this background, the following code becomes fairly clear. This is a brief excerpt of the AF_DCMotor constructor, which is initializing speed control for motor 1:

```
 // use PWM from timer2A
 TCCR2A |= _BV(COM2A1) | _BV(WGM20) | _BV(WGM21); // fast PWM, turn on oc0
 TCCR2B = freq & 0x7;
 OCR2A = 0;
 DDRB |= _BV(3);
break;
```

This can really only be understood in full detail by looking at the tables in the datasheet, which I don't want to reproduce here. However, here is a brief description of what these specific bit settings mean:

- WGM20 = 1, WGM21 = 1: selects the "fast PWM" mode described above.
- COM2A1 = 1, COM2A0 = 0: in fast PWM mode, causes the output to be set low when the comparison value is reached.
- OCR2A = 0: initializes the comparison value (i.e. the speed setting) to zero.
- Three low-order bits of TCCR2B; chooses the frequency of the PWM signal.
- DDRB bit 3 set high: sets the pin mode for the pin corresponding to output 2A. This is pin 11. This correspondence is defined by the ATmega chip itself and can't be changed.

TC0 has analogous control registers, just with "2" replaced by "0" in the names.

Finally, here's the body of the setSpeed() function. All it does is set the comparison register for the appropriate motor:

```
void AF_DCMotor::setSpeed(uint8_t speed) {
  switch (motornum) {
  case 1:
    OCR2A = speed; break;
  case 2:
    OCR2B = speed; break;
  case 3:
    OCR0A = speed; break;
  case 4:
    OCR0B = speed; break;
```

```
    }
}
```

One thing I had hoped to understand out of all this but haven't completely figured out is why no frequency control is supported for motors 3 and 4. TC0 has the same 3 bits for controlling the frequency that TC2 has, so it should be possible. It may have to do with the fact that TC0 and TC1 share the same prescaler, and TC1 is used to control the servos; but I'm not sure.

Conclusion

Hope this helps.

Michael Koehrsen
5/26/2009