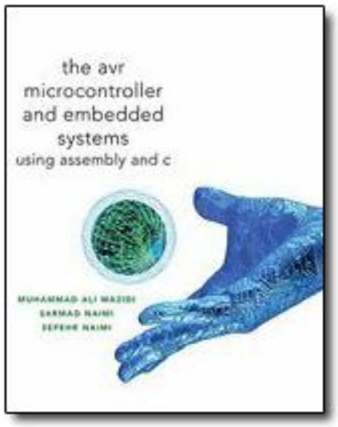
Adafruit Motor Shield - Part 1

Software Serial Peripheral Interface (SPI)

This article is the first of a two part series on the AdaFruit Motor Shield. This article focuses on a Software implementation of the Serial Peripheral Interface (SPI). The second article in the series covers Fast Pulse Width Modulation.



the avr microcontroller and embedded systems using assembly and c
MUHAMMAD ALI MAZIDI
SARMAD NAIMI
SEPEHR NAIMI

The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi
Chapter 5: Arithmetic, Logic Instructions, and Programs
    Section 5.4: Rotate and Shift Instructions and Data Serialization
Chapter 7: AVR Programming in C
    Section 7.5 Data Serialization in C
Chapter 11: AVR Serial Port Programming in Assembly and C
Chapter 17: SPI Protocol and MAX7221 Display Interfacing
Section 17.1 SPI Bus Protocol

Features

8-bit AVR® Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash

ATmega48PA
ATmega88PA
ATmega168PA
ATmega328P

ATMEL 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash
http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdfasd Chapter 21 “2-wire Serial Interface”

Table of Contents

References

1. ATMEL 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdfasd Chapter 18 "SPI - Serial Peripheral Interface"
2. ATmega328P Serial Communications (located in the EE346 Lectures folder)
3. Adafruit Motor Shield Arduino User Guide pages 4 and 17

Understanding the Adafruit Motor Shield

This document is based on an article on the Adafruit Motor Shield written by Michael Koehrsen. Original material is used by permission.

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.

Addendum

When Adafruit initially wrote the C++ code for the Motor Shield, they did it from the perspective of an AVR microcontroller. They later updated the code to take full advantage of the Arduino sketch language, leaving the original C++ code as comments. The initial draft of this article was written to explain this earlier code version. This is fortuitous, because the Arduino sketch language adds one more layer of abstraction, which is removed by studying the comments and not the new sketch version.

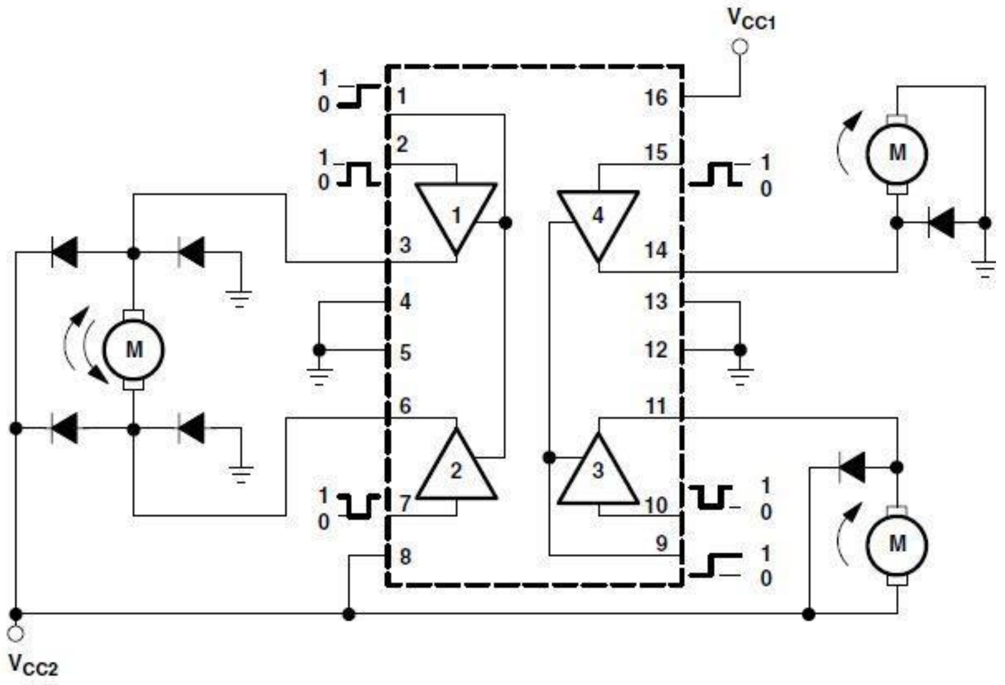
It is unfortunate that Adafruit was unable to use the Serial Peripheral Interface (SPI) subsystem of the ATmega328P, opting instead to completely mimic its function in software. The problem may have resulted from a reluctance on Adafruit's part to lose the pulse-width-modulation output PWM OC2A on pin 23 (PB3), multiplexed with MOSI.

Basic DC Motor Control

Let's set aside speed control (Part 2) 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. As shown in Figure 1, each IC has four digital inputs which control two DC 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.



NOTE: Output diodes are internal in L293D.

Figure 1 L293D dual H-bridge motor driver IC

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 (Figure 2). 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:

```
/*                      IC3 bit (source: 74HC595 datasheet) */
#define MOTOR1_A    // QC   2
#define MOTOR1_B    // QD   3
#define MOTOR2_A    // QB   1
#define MOTOR2_B    // QE   4
#define MOTOR4_A    // QF   5
#define MOTOR4_B    // QH   7
#define MOTOR3_A    // QA   0
#define MOTOR3_B    // QG   6
```

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:

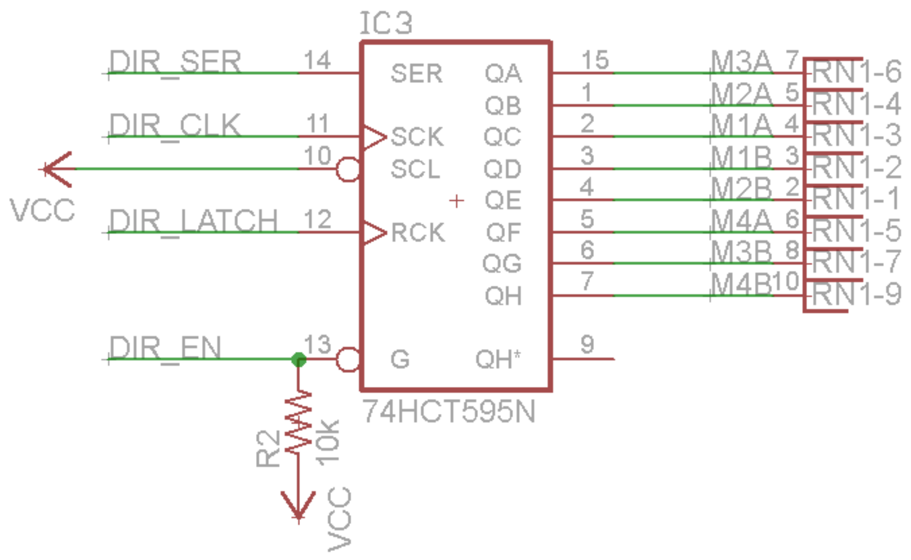


Figure 2 74HC595

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, pin 15. 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.

The pins shown on the left in Figure 2 "74HC595" are the four input pins plus reset and output enable. Adafruit's naming convention is at variance with that provided in the 74HCT595N 8-bit shift register Philips datasheet. Here is a cross-reference table

Pin	Adafruit	74HC595	Description
14	DIR_SER	DS	serial data input
11	DIR_CLK	SH_CP	shift register clock input
12	DIR_LATCH	ST_CP	storage register clock input
13	DIR_EN	/OE	output enable (active LOW)
10	SCL	/MR	master reset (active LOW) *wired to Vcc*

It takes some study of the 74HC595 datasheet to understand the inputs, so here's a Logic Diagram (Figure 6 in the Datasheet) followed by brief description of each control pint. If you do not understand how a shift register works, here is a nice Serial-in, parallel-out shift register tutorial.

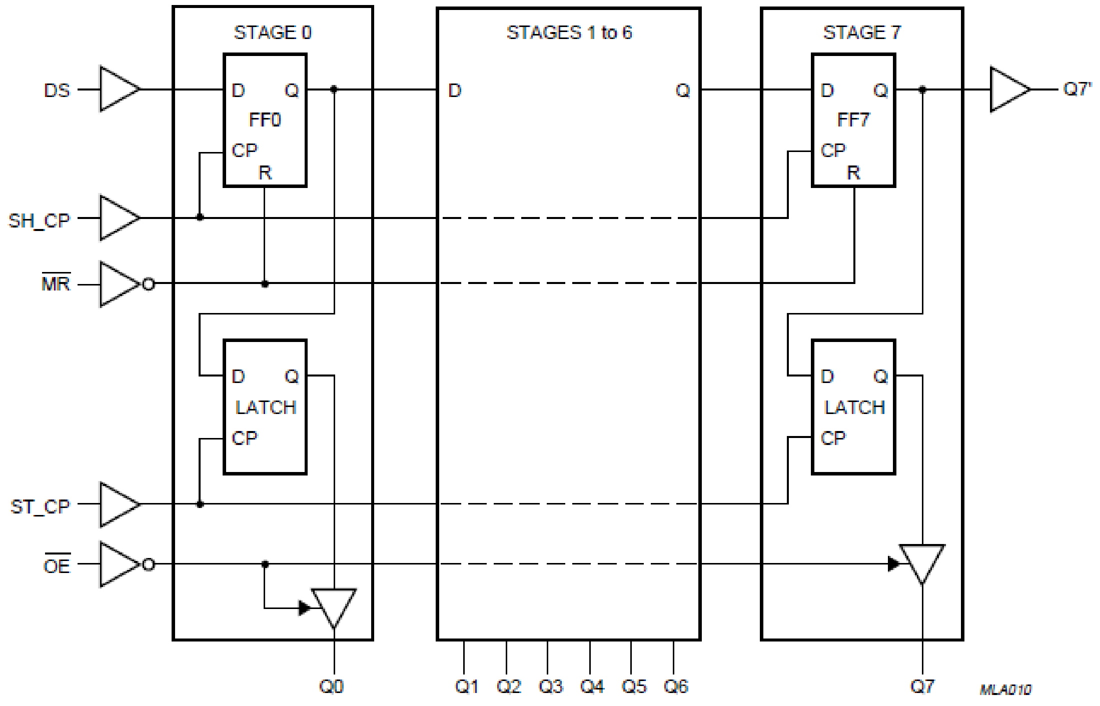


Figure 3 74HC595 Logic Diagram

- DIR_EN (/OE) - The output enable pin, (/ = 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 (DS) - the serial data input.
- DIR_CLK (SH_CP) - A rising edge on this input causes the data bit to be shifted in and all bits shifted internally.
- DIR_LATCH (ST_CP) - 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 Port Registers for a non-technical description of the general I/O ports of the ATmega328P.

To go between the ATmega328P and Arduino naming conventions use the Arduino 2009 schematic and Figure 3.

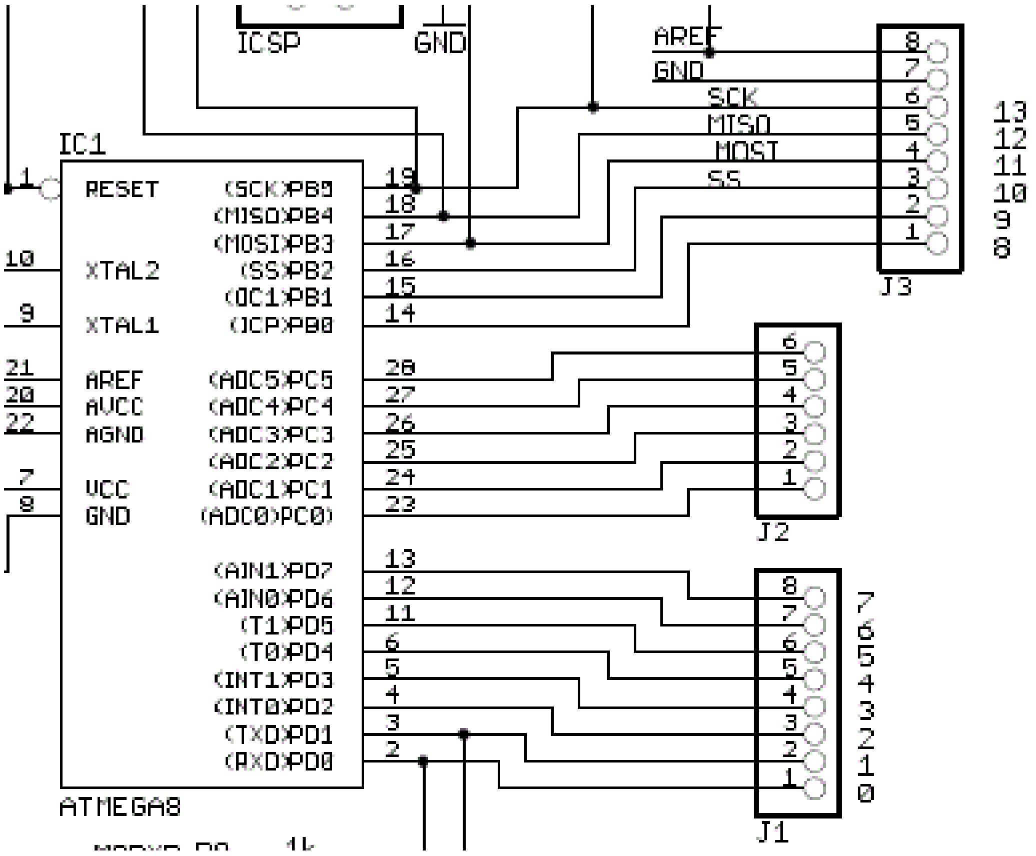


Figure 4   ATmega 328P Pin-out to the Arduino Duemilanove Digital and Analog Pin-out

Here are the salient points:

- PORTD bits 0 to 7 map to Arduino Digital pins 0 to 7, but you should never modify bits 0 and 1. These pins are reserved for serial communications (TXD, RXD).
- PORTB bits 0 to 5 map to digital pins 8 to 13; the two high-order bits 6 and 7 are wired to the crystal oscillator inputs XTAL1 and XTAL2 respectively and are therefore not available.
- PORTC bits 0 to 5 map to analog pins 1 to 6; PortC bit 7 is not supported by the ATmega328P. The Adafruit motor shield does not use these I/O bits, but they have made them available to us. These are great for adding sensors to your project. The pins are wired to Jumper JP5 as shown in the schematic and Figure 5. Jumper JP2 in Figure 3, is incorrectly labeled and should be J2 as shown in Figure 4. You can find them on your board labeled A0-5.

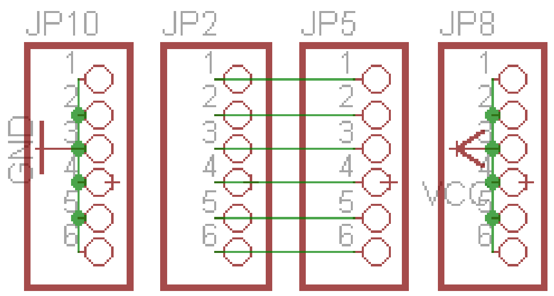
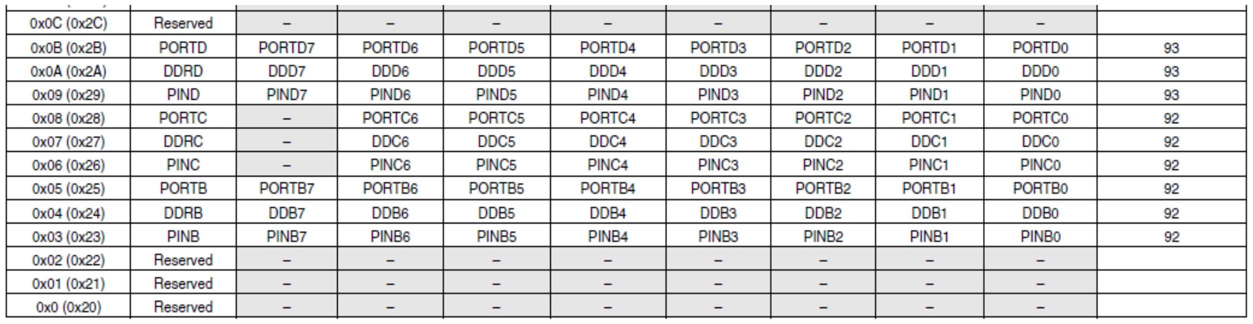


Figure 5 AdaFruit Motor Shield Jumper J2 (incorrectly labeled JP2)

For a complete cross-reference table for mapping the *tower of babel* names used by Atmel, Arduino, Adafruit, and Philips Semiconductor see Table 2.1 Microcontroller Resource Map - Motors our *Reference System Document*.

There are three registers corresponding to each port as described in the ATmega328P Datasheet and Peripherals AVR. Each Port register has a unique address as defined in the iom328p.h C header file (arduino-00nn\hardware\tools\avr\avr\include\avr\, where nn = the version number) and shown in Figure 6.



0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	93
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	93
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	93
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	92
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	92
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	92
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	92
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	92
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	92
0x02 (0x22)	Reserved	–	–	–	–	–	–	–	–	
0x01 (0x21)	Reserved	–	–	–	–	–	–	–	–	
0x0 (0x20)	Reserved	–	–	–	–	–	–	–	–	

Figure 6 ATmega328P General I/O Port Addresses

Here are the salient points:

- DDRB/DDRD/DDRC - these are hardware registers 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/PORTC - If you assign a one to them, it's like doing a `digitalWrite` on each pin.
- PINB/PIND/PINC - if you read one of these, it's like doing a `digitalRead` on each pin in the port. You should think of this Port as read only. Writing to this port is possible but not recommended for the novice. The inquisitive types can find out what will happen by reading Section 13 "I/O-Ports" in the ATmega328P datasheet (you will be surprised when you find the answer).

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/PIND as opposed to `digitalWrite` and `digitalRead`.

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

// Arduino pin names
#define MOTORLATCH 12
#define MOTORCLK 4
#define MOTORENABLE 7
#define MOTORDATA 8
```

Clearly these five groups of #define statements correspond one-to-one with the inputs to the shift register, and specifically they define the relationship between ATmega328P Port bits, Arduino pins, and shift register inputs (see Reference System Desgin Table 2.1 Microcontroller Resource Map - Motors for details) .

Table 1   AdaFruit Motor Shield to ATmega328P to Arduino

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:

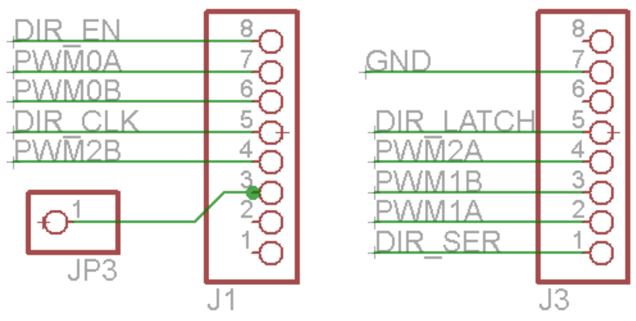


Figure 7 Adafruit Motor Shield signals to Arduino Duemilanove Jumper Pin-out

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 *or* Software SPI

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 (arduino-00nn\hardware\libraries\AFMotor\, where nn = the version number); this instance is used by the classes **`AF_DCMotor`** and **`AF_Stepper`**.

AFMotorController declares two member functions (i.e., methods) **`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:

```
/*
      Send data located in 8-bit variable latch_state to
      the 74HC595 on the Motor Shield.
*/
void AFMotorController::latch_tx(void) {
  uint8_t i;

  //LATCH_PORT &= ~_BV(LATCH);
  digitalWrite(MOTORLATCH, LOW);     // - Output register clock low

  //SER_PORT &= ~_BV(SER);
  digitalWrite(MOTORDATA, LOW);      // - Serial data bit = 0

  for (i=0; i<8; i++) {              // - Shift out 8-bits
      //CLK_PORT &= ~_BV(CLK);
      digitalWrite(MOTORCLK, LOW);   //     - Shift clock low

      if (latch_state & _BV(7-i)) { //     - Is current bit of
      //SER_PORT |= _BV(SER);             latch_state == 1
      digitalWrite(MOTORDATA, HIGH); //    -   Yes, serial data bit = 1
      } else {
      //SER_PORT &= ~_BV(SER);
      digitalWrite(MOTORDATA, LOW);  //    -   No, serial data bit = 0
      }
      //CLK_PORT |= _BV(CLK);
      digitalWrite(MOTORCLK, HIGH); //     - Shift clock high, rising edge
  }                                  //     shift bit into shift register
  //LATCH_PORT |= _BV(LATCH);
  digitalWrite(MOTORLATCH, HIGH);    // - Output register clock high, rising
}                                    //   edge sends the stored bits to the
                                     //   output register.
```

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))
```

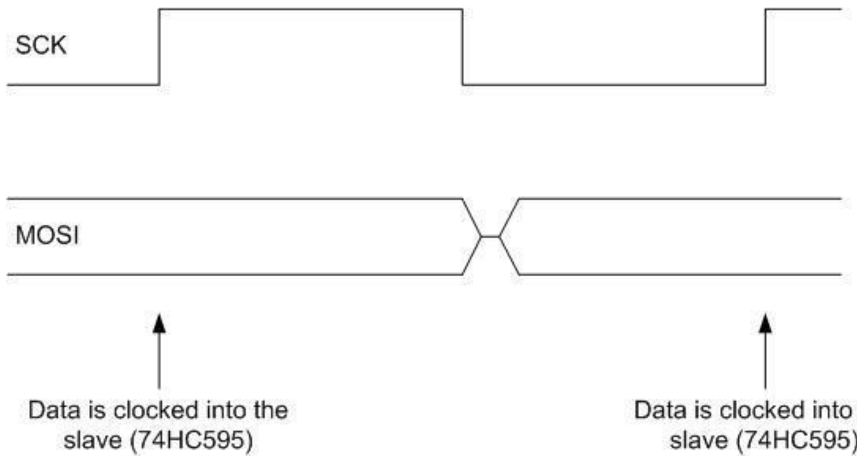
Knowing that, you can easily work out how the following two statements set a single bit of SER_PORT to high and low respectively. The same idiom is used everywhere in this code so it's worthwhile to think through it if you're not used to doing bit manipulations.

```
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).

```
/*
    Configure DDR Registers B and D bits assigned to
    the input of the 74HC595 on the Motor Shield. Output
    all zeros and enable outputs.
*/

void AFMotorController::enable(void) {
  // setup the latch
  /*
  LATCH_DDR |= _BV(LATCH);
  ENABLE_DDR |= _BV(ENABLE);
  CLK_DDR |= _BV(CLK);
  SER_DDR |= _BV(SER);
  */
  pinMode(MOTORLATCH, OUTPUT);
  pinMode(MOTORENABLE, OUTPUT);
  pinMode(MOTORDATA, OUTPUT);
  pinMode(MOTORCLK, OUTPUT);

  latch_state = 0;

  latch_tx();  // "reset"

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

The `for` loop implements the following waveform programmatically. You may recognize this waveform. it is the same one in the SPI Serial Communications document.


SCK
MOSI
Data is clocked into the slave (74HC595)
Data is clocked into the slave (74HC595)

AF_DCMotor *or* Forward and Backward

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;
  }
}
```