Get Your Motors Runnin'

The very first experience I had with a hobby servo involved a 555 timer. I had a friend that was deep into RC aviation. One day, we were sitting in his shop working on a plane and I noticed that he kept picking up the transmitter to test his servo linkage hookups. Not only did he have to fire up the transmitter, he had to make sure the newly installed servo was attached to the receiver. It looked painful to me. So, I suggested that I create a small box that could drive the servo directly without the need of a transmitter and receiver. After some intense reading (no Internet in those days), I figured out how to make a 555 timer and a single-axis potentiometer-based joystick do the job of an RC transmitter and receiver.

By Fred Eady

Post comments on this article and find any associated files and/or downloads at **www.servomagazine.com/index.php?/magazine/article/november2013_Eady**.

Digilent, Inc.
Digilent Motor Shield
Uno32
MPIDE
Gear Motor
www.digilentinc.com

When an RC servo is forced to move, it makes a very distinctive sound. As long as there are humans walking the earth, there will be a need for wheels, motors, and RC servos. With that, this month's discussion will revolve around driving RC servos and DC motors using a Digilent motor shield mounted on a Uno32.

Uno32 Revival

The Uno32 smiling in **Photo 1** had its original factory bootloader overwritten. In that the Digilent motor shield was originally designed to run under the control of firmware generated by MPIDE, it might be a good idea to restore the Uno32's factory bootloader image. The re-image process is pain free.

First, I downloaded the Uno32 bootloader image from the Digilent website. The second step involved importing the downloaded bootloader hex image into MPLAB. Once the bootloader hex file was imported, I used a PICkit 3 to Flash the replacement factory bootloader image. To verify that the bootloader Flash completed successfully, I used MPIDE

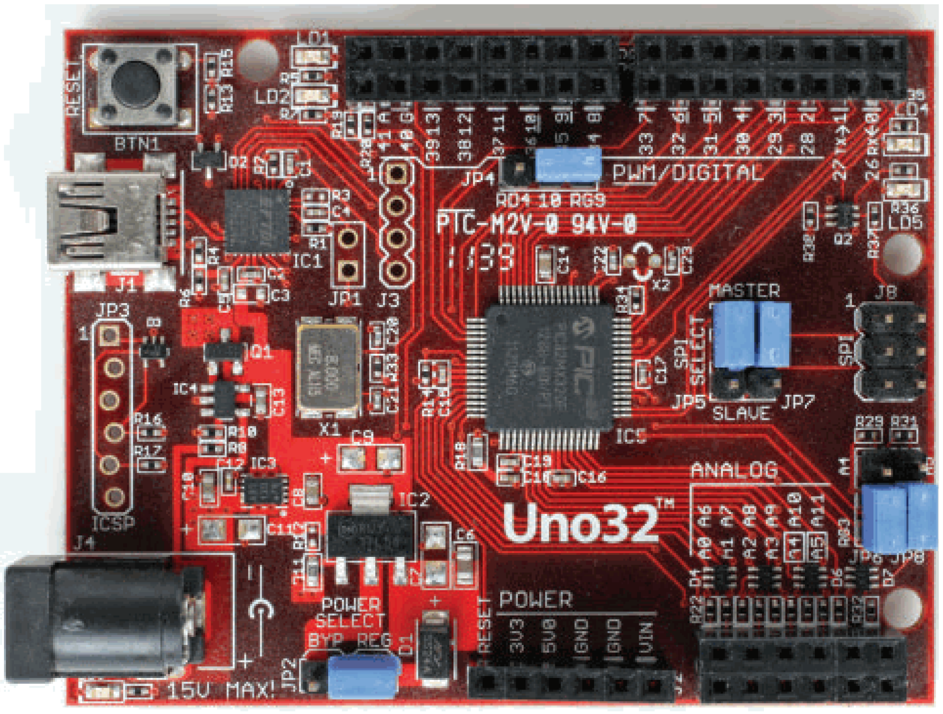


Photo 1. The name pretty much says it all. The Uno32 is based on a Microchip PIC32MX microcontroller and is capable of running many of the original Arduino sketches.

to load and execute a simple LED blinky program. The Uno32 is back in business.

The Digilent Motor Shield

The key to understanding how to use the motor shield pictured in **Photo 2** is to have a firm grasp on the hardware components that make it up. Please reference **Photo 2** and the motor shield schematics as we walk through its electrical subsystems.

DC Motor Drive Subsystem

As you can see in **Schematic 1**, the motor shield is based on the Texas Instruments DRV8833 dual H-bridge motor driver. The DRV8833 is capable of driving one stepper motor or two DC motors. You can see this a bit more clearly in **Figure 1**.

Basically, the output pins AOUTx and BOUTx logically follow the AINx and BINx input pins. The logical relationship that exists between the DRV8833's input and output pins is illustrated in **Figure 2**. The reverse and forward modes are what one would expect.

Let's talk about the coast and brake modes. Coasting occurs when the H-bridge is shut down. In coast mode, the recirculation current is allowed to flow through the MOSFET body diodes. The MOSFET body diodes are the reverse biased diodes located across each MOSFET's drain and source. Brake (slow decay) mode does not allow the recirculation current to flow through the MOSFET body diodes by shorting the motor winding.

The motor speed can be

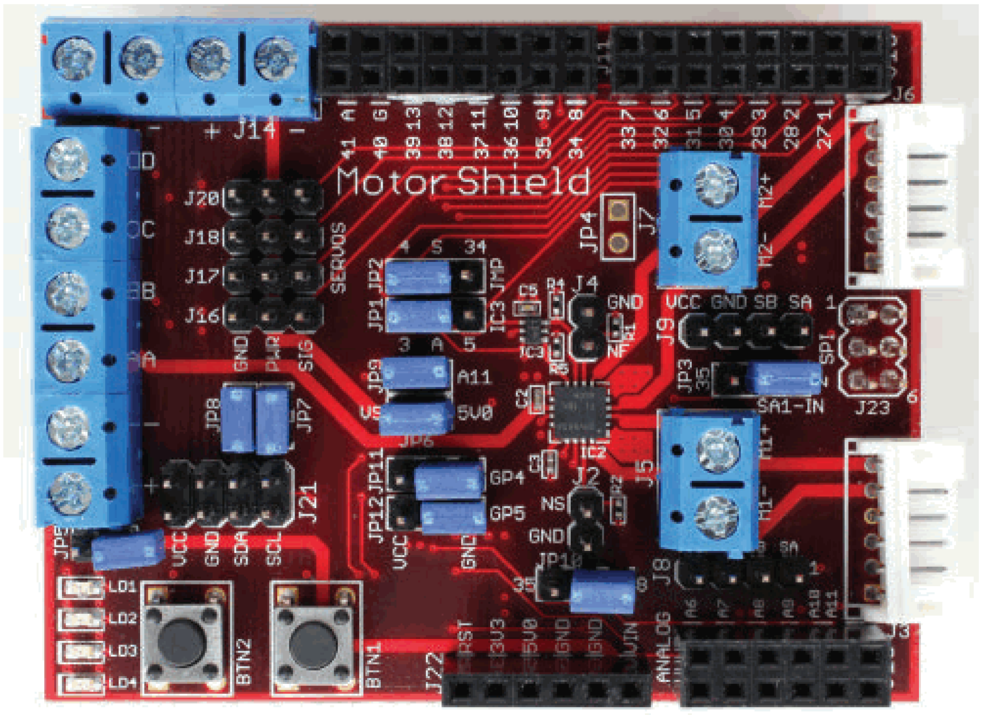
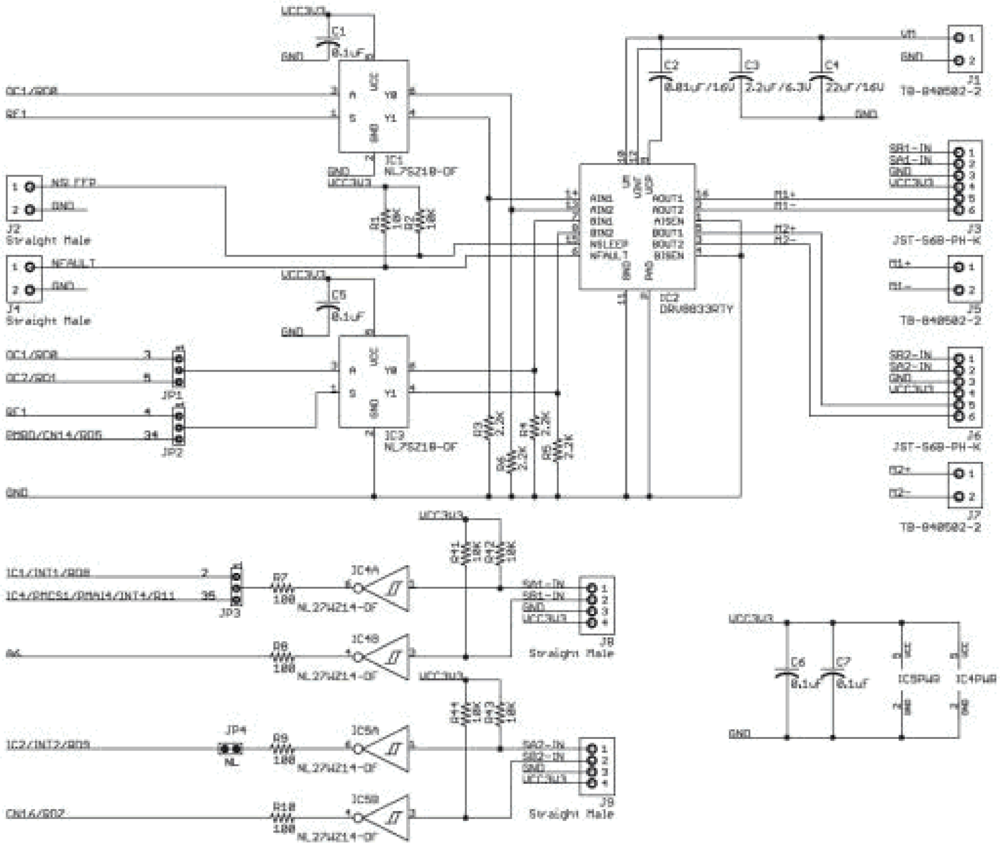


Photo 2. The Digilent motor shield can drive up to two DC motors or one stepper motor. There are also pins to accommodate up to four RC servos. An I2C expander gives the user an additional two pushbuttons and four LEDs.


Schematic 1. This was obviously designed to accommodate all of the features of the Digilent gear motors. However, it is a flexible design that is also capable of driving just about any small two-wire DC motor.

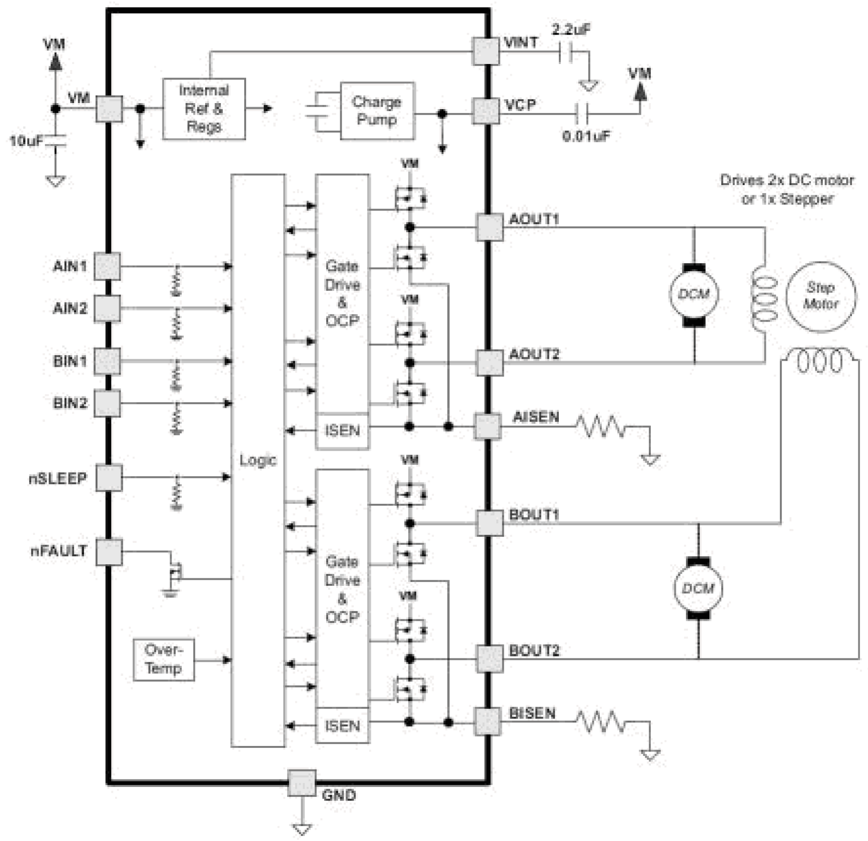


Figure 1. There are plenty of words describing the DRV8833 in the datasheet. Once again, a picture is worth 1,000 of them.

Input		Output	
S	A	Y_0	Y_1
L	L	L	Z
L	H	H	Z
H	L	Z	L
H	H	Z	H

Figure 4. The automatic high impedance state change allows the resistors to force their associated demultiplexer input pin logically low.

controlled by applying a pulse width modulation (PWM) signal to the DRV8833 input pins. Naturally, the PWM mode also allows the attached motor or motors to be reversed. The PWM logic table for the DRV8833 is contained within **Figure 3**.

xIN1	xIN2	xOUT1	xOUT2	FUNCTION
0	0	Z	Z	Coast/fast decay
0	1	L	H	Reverse
1	0	H	L	Forward
1	1	L	L	Brake/slow decay

Figure 2. The relationship between the input pins and output pins is rather obvious. However, it may not be intuitively obvious that fast decay disables the H-bridge and allows the recirculation current to flow through the MOSFET body diodes. Slow decay mode shorts the motor winding.

xIN1	xIN2	FUNCTION
PWM	0	Forward PWM, fast decay
1	PWM	Forward PWM, slow decay
0	PWM	Reverse PWM, fast decay
PWM	1	Reverse PWM, slow decay

Figure 3. Now that you know what the decay modes consist of, this too becomes a logical table of operations.

Let's shift our attention to **Schematic 1**. The DRV8833 AINx and BINx inputs are all tied logically low via resistors R3-R6. The DRV8833 AINx and BINx pins are being fed from a pair of 1:2 demultiplexers. The logic level at the A input of each multiplexer is passed to the Y0 or Y1 demultiplexer output, depending on the logical state of the select pin. The Yx output pin that is not selected reverts to a high impedance state. In that the DRV8833 inputs are pulled logically low, it is imperative that the deselected demultiplexer output not influence the DRV8833 input pin. The NL7SZ18 1:2 demultiplexer truth table is shown in **Figure 4**.

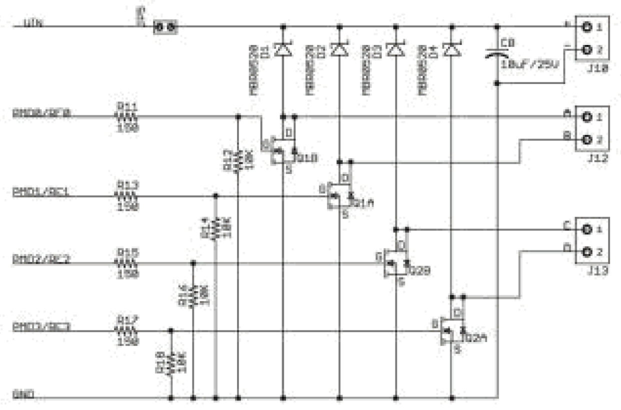
We can close out our DRV8833 input circuitry examination by observing that shorting J2 will put the DRV8833 to sleep and disable the H-bridge circuitry. We can also say with certainty that if we manage to overcurrent, overheat, or undervoltage the DRV8833 outputs, an active low fault signal will be presented at pin 1 of J4. If the fault continues to occur while driving a single motor with a single output, we have the option to parallel the DRV8833 outputs which will increase its output current handling capability. The DRV8833 can drive motors in the voltage range of 2.7 to 10.8 VDC. The motor drive voltage must be supplied at J1.

We can also put a cap on the walk through of the DRV8833 output circuitry. The Digilent gear motor resting in **Photo 3** generates those SAx quadrature encoder signals you see at J3 and J6. The quadrature-formatted encoder signals are used to indicate speed and direction of the motor shaft. The raw quadrature-encoded signals are buffered by IC4 and IC5. The quadrature encoder signals are optional, and a standard two-wire DC motor can be driven without the assistance of the SAx quadrature encoder signals using J5 and J7.

As you can see, the DRV8833's AISEN and BISEN current sense pins are both grounded. This indicates that the motor shield design is not taking advantage of the DRV8833's current monitoring pins.

Stepper Motor Driver Hardware

This subsystem does not depend on the DRV8833.



Schematic 2. There's not much to say about this circuit. However, if you don't drive a stepper motor with your motor shield, you can use the MOSFETs as high current solid-state switches.

Photo 3. This Digilent gear motor interfaces directly to the motor shield via J3 or J6. A pair of Hall-effect sensors produces a quadrature signal as the motor shaft rotates.

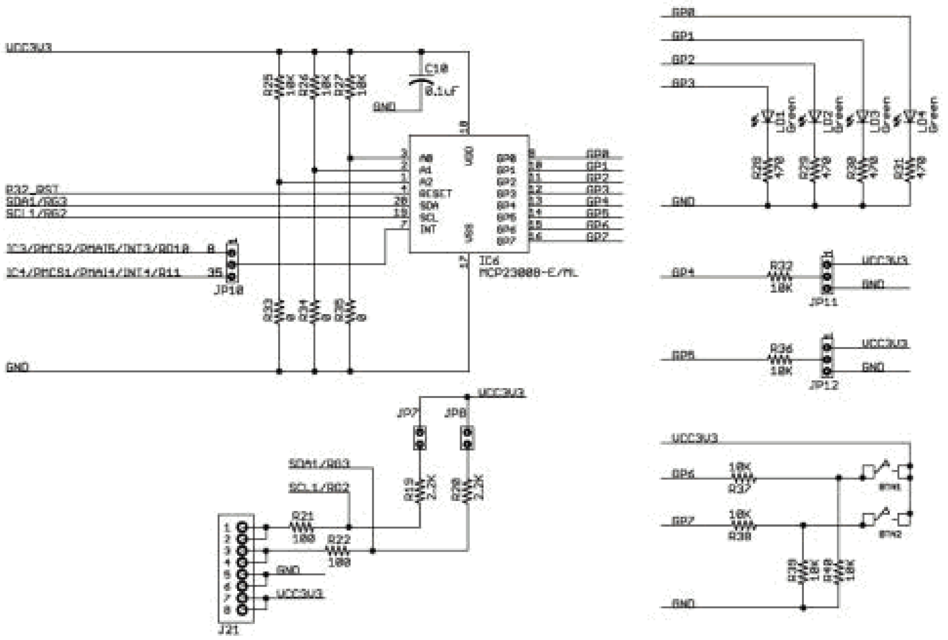
Instead, the stepper motor drive electronics consist of four open drain MOSFETs and four associated steering diodes. The stepper motor driver hardware is graphically depicted in **Schematic 2**. The quad of open drain MOSFETs can also be used as independent solid-state switches. The stepper motor driver subsystem can be powered externally by removing JP5 and supplying power via J10.

Servo Drive Subsystem

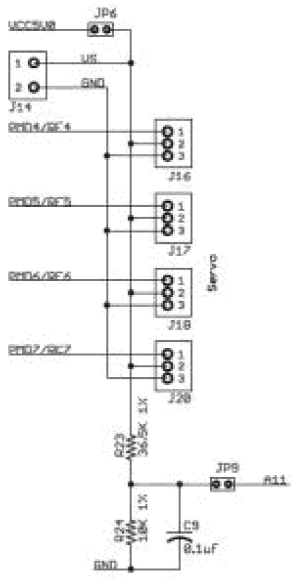
The servo drive subsystem consists of a couple resistors and a capacitor. Hobby servos are driven with a logic level signal. Thus, the servo drive motor is the only power consuming element. This is reflected in **Schematic 3**, which details four servo positions. Resistors R23 and R24 form a simple voltage divider that allows the servo power source voltage to be monitored by the host microcontroller. Like the stepper motor driver subsystem, the servo drive subsystem can be powered externally or from the host 5 VDC rail.

I²C Expander Subsystem

This is a very interesting part of the motor shield. The center of this subsystem is a Microchip MCP23008 I²C expander. The expander IC you see in **Schematic 4** is nothing more than an eight-bit I/O expander that is



Schematic 4. The motor shield takes advantage of the Uno32's I2C bus and employs the services of the MCP23008 to add LEDs, pushbuttons, and a pair of user inputs.



Schematic 3. Again, there is not much to discuss. The magic that positions each servo is performed by the host microcontroller.

configured and controlled using the I²C protocol. The MCP23008's eight I/O pins are configured as inputs on high nibble and output on the low nibble. The low nibble drives a quad of LEDs. Pushbuttons are attached to the two most significant bits of the high nibble. GP4 and GP5 are dedicated to jumper switches that provide a logical high or logical low input.

Digilent Motor Shield Firmware

The motor shield can be driven using the SoftPWMServo and Wire libraries. However, there is also a dedicated MotorShield library that takes the pain out of communicating with the MCP23008. Let's see what it takes to spin some motor shafts and blink some LEDs using the I²C bus.

Driving Miss DC Gear Motor

Referencing the chipKIT *Motor Shield Reference Manual*, we find that the J3 motor interface's Arduino enable pin (Enable1) is Uno32 pin 3. The associated direction pin (Direction1) is assigned to pin 4. We will use the SoftPWMServo library to place a PWM signal on the Enable1 pin:

```
#include <SoftPWMServo.h>

// Uno32 Enable and Direction Pin Assignments
const int pEnable1 = 3;
const int pDirection1 = 4;
```

The motor control logic behind the enable and direction versus PWM signals is contained within **Figure 5**. We'll need a place holder for the PWM value, and we'll also need a logical switch to toggle the Direction1 pin:

```
int pwmVal;           // PWM value
bool bToggle;         // forward/reverse switch
```

Okay. Now that all of our variables and pins are defined, let's go ahead and initialize them. While we're at it, let's initiate a serial link so we can use the serial monitor as a debugger:

```
void setup()
{
     pinMode(pEnable1, OUTPUT);
     pinMode(pDirection1, OUTPUT);
     bToggle = true;
     Serial.begin(9600);
}
```

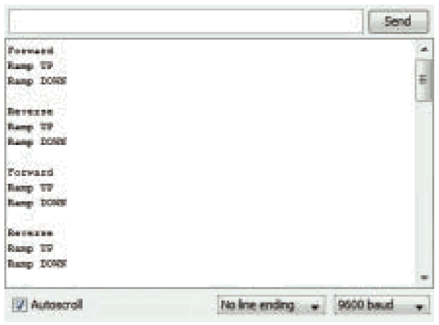
The plan is to accelerate the gear motor, decelerate it, change directions, and repeat the whole process again:

```
void loop()
{
     switch(bToggle)
     {
       case true:
         digitalWrite(pDirection1, LOW);
                             // forward
         Serial.println("Forward");
         delay(1000);
       break;
       case false:
         digitalWrite(pDirection1, HIGH);
                             // reverse
         Serial.println("Reverse");
         delay(1000);
       break;
     }
      Serial.println("Ramp UP");
     for(pwmVal= 0x00;pwmVal < 0x0100;pwmVal++)
     {
        SoftPWMServoPWMWrite(pEnable1, pwmVal);
                             // send PWM value
       delay(50);
     }
       Serial.println("Ramp DOWN\r\n");
     for(pwmVal=0xFF;pwmVal > 0;pwmVal—)
     {
       SoftPWMServoPWMWrite(pEnable1, pwmVal);
                             // send PWM value
       delay(50);
     }
     bToggle = !bToggle;
}
```

Our little motor twister program checks the condition of the *bToggle* variable and determines how to set the logical value of the Direction1 pin. The gear motor is then ramped up and ramped down in the selected direction. The *bToggle* variable gets inverted and we perform the ramp up/ramp down in the opposite direction. To check things along the way, the direction and ramp status is sent out to the serial monitor which is captured in **Screenshot 1**. Power consumption with the motor running is 70 mA.

Positioning the Servo

Again referencing the motor shield manual, we find



```
Forward
Ramp UP
Ramp DOWN

Reverse
Ramp UP
Ramp DOWN

Forward
Ramp UP
Ramp DOWN

Reverse
Ramp UP
Ramp DOWN
```

Screenshot 1. **This display serves two purposes. I use it to notify me of the progress of the program and to verify that the motor was really doing what I told it to do.**

DIR1	EN1	Result
0	0	Stop
0	1/PWM	Forward
1	0	Stop
1	1/PWM	Reverse
DIR2	**EN2**	**Result**
0	0	Stop
0	1/PWM	Forward
1	0	Stop
1	1/PWM	Reverse

Figure 5. This table is valid for steady state and PWM motor control.

that Servo1 and Servo2 are accessed via Uno32 Arduino pins 30 and 31, respectively. Oh, heck. We'll enumerate all four and we'll reuse the SoftPWMServo library:

```
#include <SoftPWMServo.h>

const int servo1 = 30;  // Servo 1
const int servo2 = 31;  // Servo 2
const int servo3 = 30;  // Servo 3
const int servo4 = 31;  // Servo 4
```

The library function call *SoftPWMServoServoWrite(pin, position)* will move the servo motor associated with the Arduino pin number to a position represented by microseconds. The extent of a typical hobby servo is 1,000 µS to 2,000 µS, with center at 1,500 µS. So, to center Servo1, we would substitute 30 for pin and 1500 (1.5 mS) for position. Typing out the servo library call could get to be a pain. To keep our fingers sane, let's wrap the servo movement library call into a couple of C functions:

```
static void centerServo(int servonum)
{
     SoftPWMServoServoWrite(servonum, 1500);
}

static void moveServo(int servonum, int pos)
{
      SoftPWMServoServoWrite(servonum, pos);
}
```

As you can see, the *centerServo* function simply commands the servo to center its shoe. If we need to arbitrarily move the servo shoe, we place a call to the *moveServo* function. Let's go ahead and perform the servo Arduino pins setup:

```
void setup()
{
  pinMode(servo1,OUTPUT);
  pinMode(servo2,OUTPUT);
  pinMode(servo3,OUTPUT);
  pinMode(servo4,OUTPUT);
  Serial.begin(9600);
}
```

Once again, we'll fire up the MPIDE serial monitor for the same reasons we did before. The code will be very easy to read since our servo function calls are self-documenting:

```
void loop()
{
  centerServo(servo1);
  Serial.println("Centered");
  delay(2000);
  moveServo(servo1,1000);
  Serial.println("1000");
  delay(2000);
  centerServo(servo1);
  Serial.println("Centered");
  delay(2000);

  moveServo(servo1,2000);
  Serial.println("2000");
  delay(2000);
}
```

The *delay* function calls provide way more time than required for the servo shoe to complete the predetermined moves that are programmed in. The serial monitor output for our servo application is shown in **Screenshot 2**.

Mastering the I²C Expander

The I²C expander IC can be manipulated with the Arduino wire library. When you get your homebrew wire code, you will come to the realization that you have spent a bunch of time reinventing the MotorShield library code.

The format of an I²C exchange between the Uno32 and the MCP23008 begins with a start bit, followed by a control byte. The control byte encapsulates the slave address and the R/W (Read/Write) bit. The most significant seven bits of the control byte contain the slave address.

Take another look at **Schematic 4**. The MCP23008's address lines A0, A1, and A2 are all pulled to ground with ZeroΩ resistors. Thus, one would say the I²C address of the MCP23008 is zero (000). Wrong!

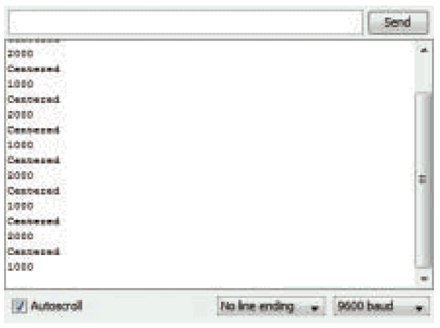
The MCP23008 datasheet tells us that the MCP23008 address bits are located in the three least significant bits of the seven-bit slave address field. The upper four bits of the MCP23008 slave address are 0100. This is a bit confusing as a discussion. So, let's map it out:

```
S 0100 A2 A1 A0 R/W
Where:
       S = Start bit
       Slave Address = 0100000
       R/W = Read/Write bit
The Slave Address is partitioned as 010 0000 or
0x20.
```

The 0x20 gets you to the MCP23008's front door. The next thing you better have is the room address. The "room" address is really one of 11 MCP23008 register addresses. Once you enter the right room, you can deliver that pizza in the bag you're carrying. Here's what that front door, room, and pizza look like in the *MotorShield.h* file:

```
// unique I2C device address
 #define DEVADDR      0x20

 //Register Addresses
 #define IODIR         0x00
       // Data direction register (setting pins
       // for input/output)
 #define IPOL          0x01
 #define GPINTEN       0x02
 #define DEFVAL        0x03
 #define INTCON        0x04
 #define IOCON         0x05
```



```
2000
Centered
1000
Centered
2000
Centered
1000
Centered
2000
Centered
1000
Centered
2000
Centered
1000
```

Screenshot 2. I've moved my share of servo shoes. This has to be the easiest way to do it that I've seen so far.

```
 #define GPPU 0x06
 #define INTF        0x07
 #define INTCAP      0x08
 #define GPIO        0x09
       // the main data port for reading and
       // writing
 #define OLAT 0x0A
```

The pizza is the data you load into the appropriate MCP23008 register. Here's a typical write register sequence that targets the GPIO register:

```
       Wire.beginTransmission(DEVADDR);
       // transmit to motor shield
       Wire.send(GPIO);
       // select the GPIO register
       Wire.send(value);
       // data to load into the GPIO register
       Wire.endTransmission();
       // stop transmitting
```

The MotorShield library contains the following publically accessible functions:

```
void begin(void)
void writeLEDs(byte value)
byte readInputs(void)
void readButtons(bool* btn1, bool* btn2)
```

The GPIO register code snippet is actually part of the *writeLEDs* library call. To use the library functions, we must first instantiate a MotorShield class. Here is the class code that lies within the MotorShield library:

```
/*******************
 * MotorShield Class
 ******************/

class MotorShield
{
       public:
       void begin(void);
       void writeLEDs(byte value);
       byte readInputs(void);
       void readButtons(bool* btn1, bool* btn2);

       private:
       void reorder_LEDs(byte* value);
};
```

Here's how to instantiate a MotorShield class:

```
// Wire.h must be included when using the
// MotorShield.h library
#include <Wire.h>
#include <MotorShield.h>

// Create an instance of MotorShield called
// extender
MotorShield extender;
```

The next step is to finish up our variable declarations and kick off *extender*:

```
byte x;
bool btn1, btn2;

void setup()
{
    extender.begin();    // initialize the I/O
                         // extender
}
```

The *setup* function code is pretty vague. A peek under the hood will clear things up:

```
void MotorShield::begin(void)
{
       Wire.begin();
       // join the I2C bus as master

       Wire.beginTransmission(DEVADDR);
       // transmit to motor shield
       Wire.send(IODIR);
       // select the data direction register
       Wire.send(0xF0);
       // configure high half of I/O as inputs,
       // and the low half as outputs
       Wire.endTransmission();
       // stop transmitting

}
```

Setting the MCP23008's IODIR register is analogous to setting PIC I/O pins as inputs or outputs.

Let's write an application that reads the pushbuttons and illuminates the motor shield's onboard LEDs accordingly. Here's the truth table we will code by:

```
btn1 depressed - btn2 released = LED1 ON
btn1 released - btn2 depressed = LED2 ON
btn1 depressed - btn2 depressed = LED3 ON
```

I can't show you the LEDs, but I can show you the code:

```
void loop()
{
  // readButtons sets the bool values, 1 for
  // pressed button, 0 for unpressed button
  extender.readButtons(&btn1, &btn2);

  x = 0x00;

  if (btn1 && !btn2)
    x = x | 0x01;          //illuminate LED1
  if (!btn1 && btn2)
    x = x | 0x02;          //illuminate LED2
  if (btn1 && btn2)
    x = x | 0x04;          //illuminate LED3

  // writes the binary value of x (first 4 bits)
  // to the LEDs
  extender.writeLEDs(x);
}
```

Double-check my code when you get your own motor shield.

The Possibilities

You can drive a pair of DC motors, a stepper motor, and control up to four servos simultaneously. You can use the motor shield pushbuttons to activate or deactivate a motor or servo. The state of a motor, servo, or LED can be altered according to the logic levels of the jumper-switch inputs. Extend the I^{2}C bus to control external I^{2}C slave devices via J21. Drive small relays using the stepper motor MOSFETs. Or, just have fun writing code to make stuff move and blink LEDs. **SV**