Interfacing a Servo to the ATmega16

Learning Objectives:
After successfully completing this lab, students will be able to:
- Describe how a radio control (RC) servo works
- Interface a RC servo to a microcontroller
- Use the ATmega16 to control the movement of a servo

Components:

<table>
<thead>
<tr>
<th>Qty</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ATmega16 microcontroller with STK 500 development board, serial port cable</td>
</tr>
<tr>
<td>1</td>
<td>Futaba FP-S148 servo or similar</td>
</tr>
</tbody>
</table>

Introduction:
In this lab you will explore the workings of a radio control (RC) servo and how the ATmega16 can be used to control one. An RC servo consists of a dc motor, gear train, potentiometer, and some control circuitry all mounted compactly in a case. RC servos are commonly used in radio-controlled cars, airplanes, and boats to provide limited rotational motion to steer, move control surfaces, etc. RC servos are attractive for educational use in mechatronics, because they are relatively inexpensive (about $12-$20), they can put out about 42 oz/in of torque, they can easily be modified to produce continuous shaft rotation at relatively slow speeds, and they can easily be controlled by a microcontroller. Figure 1 shows that there are three wires, white, red, and black on the servo leading from a 3-pin female connector to the case. These carry the control signal, power, and ground return respectively.

Figure 1. RC Servo. The RC servo uses three wires: white carries the control signal, red carries power (usually 4.8 V to 6 V), and black is ground.

Figure 2 shows how an RC servo is made to rotate. The control circuitry inside the servo must receive a stream of pulses whose widths may vary between about 1 to 2 ms. These pulses must occur at intervals of about 10 to 20 ms. A potentiometer coupled to the rotation of the output shaft produces a voltage corresponding to the angle of the shaft. The control circuitry compares the “average” voltage of the control signal (i.e., by low pass filtering) with the voltage from the potentiometer, and the shaft rotates until the two voltages are the same.

Figure 2. RC Servo Operation. The RC servo needs to see a pulse-width modulated control signal in order to position the output shaft. Pulse-widths vary between approximately 1ms to 2 ms, and have a period between 10 ms to 20 ms.
**Procedure:**

In this lab, we are going to build a circuit to control the servo using the switches on the STK500.

1. As shown in Figure 3, connect the PORTA pins to the 10-pin header for the SWITCHES using a 10-pin ribbon cable jumper. Similarly, connect the PORTC pins to the 10-pin header for the LEDs. (Figure A1 in the Appendix explains how the switches and LEDs on the STK500 are wired internally.) Do not connect the servo yet!

2. Test your connections to the switches and LEDs by starting a new project in AVR Studio using the `servo_switch_control_test.c` program found on the ME 106 website. Your program should allow you to turn on the LEDs when the corresponding switch is pressed. **Explain how the if-else statement works and the significance of the ‘~’ operator.**

![STK500 figure](image)

**Figure 3. Servo control circuit.** PORTA pins are jumpered to the SWITCHES, and PORTC pins are jumpered to the LEDs on the STK500. The 5V source to power the servo can come from a VTG pin of one of the headers on the STK500.

3. After you have proven that the switches and LEDs have been wired correctly, add the servo. Connect the servo’s white wire to the STK500 board at pin PB3. The red and black wires should be connected to power and ground, and thus can be hooked up to a VTG and GND on the STK500 board. Now connect the oscilloscope to the white wire going to the servo. Don’t forget to connect the oscilloscope’s ground clip to a GND pin on the STK500. Also, we will be using RS232 communication for the next part of the lab, so connect a jumper from RS232 SPARE to pins PD0 and PD1.

4. Create a new project in AVR Studio, copy `main_servo.c`, `uart.c`, and `uart.h` from the ME 106 website into your project folder, and add these files to AVR Studio as you have done in previous labs. You are going to need to print out floating point values, so follow the instructions in the comments in `uart.h` to add the necessary information in your linker options. See Figure 4 below. Click on the Custom Options button on the left hand side of the window, click [Linker Options] in the list and paste this line in the text field. Don’t forget to press the Add button!
5. Before we continue, let’s take a look at a couple of sections of the demo program to try and get a better idea of what’s going on. In the RC Filter lab, we used a timer to generate various tones to mimic a simple piano. We did this by setting up Timer1 (a 16-bit timer) so that we could vary the period while keeping the duty cycle constant at 50%. In this lab we will use Timer0 (an 8-bit timer) to vary the duty cycle while keeping the period constant.

The timers that are included on the AVR line of microcontrollers are really multi-purpose devices that are capable of acting as counters that can count a number of external pulses, timers that can trigger a software interrupt after a certain period of time, as well as devices that generate square waves of varying duty cycles and periods. For this lab we will use a timer to generate a PWM signal in a configuration known as Phase Correct PWM. Phase Correct PWM works by incrementing a counter, which is attached to a reference clock signal until the counter reaches a maximum value (255 for Timer0). When this maximum value is reached, the counter starts counting back down to zero, and the process repeats. At every increment of the counter, the timer compares the value of the counter to a value stored in an Output Compare Register, if these values are the same it toggles an output pin on the microcontroller, that is, it changes its value from high (5V) to low (GND) or vice-versa.

Let’s take a look at the init() function in the demo program:

```c
void init()
{
    //Initialize Timer0
    TCCR0 = _BV(WGM00) | _BV(COM01) | _BV(CS02);

    //Enable output
    DDRB |= _BV(PB3);
}
```
Just like the ports and all other components of the ATmega16, the timers are controlled by registers. The registers that we need to deal with for Timer0 are: **TCCR0** (Timer Counter Control Register 0) which configures the timer, and **OCR0** (Output Compare Register 0), which holds the compare value.

We set the WGM00 bit in TCCR0, which sets the timer to Phase Correct PWM, the COM01 bit to enable the output of the PWM on pin PB3, and the CS02 bit to select the CPU clock/256 as the clock source for the timer. All of these registers and bits are discussed in greater detail in the ATmega16 datasheet.

The last thing we need to do is configure the port containing the pin which will output the PWM signal. This is done by setting the appropriate bit in the DDRB register. Note that the datasheet refers to the PWM output pin as OC0, not as PB3. Since almost all the pins have additional functionality associated with them besides standard IO, we can see which pins are associated with that functionality by looking at the “pinout” on page two of the ATmega16 datasheet, also shown below in Figure 6.

**Figure 5. ATmega16 pinout.** The pinout not only shows which pins are associated with which ports, but also what other functionality that they may provide. For example, we can see that the output compare pin for Counter/Timer0 (OC0) is on PB3, that the UART/RS232 communication (RXD and TXD) are on pins PD0 and PD1, and that the Analog to Digital Conversion (ADCx) pins are all on PORTA. *Figure from page 2 of the ATmega16 Datasheet, ©2008 Atmel Corporation.*

Next, we’ll look at part of the UART character received interrupt where we actually set the Output Compare Register:

```c
if ((input == 'a') && (OCR0 != 255))
{
    OCR0++; //Increment timer0's Output Compare Register
    printf("Period: %f/%f\tOCR0: %u/255\r\n", T*OCR0/255, T, OCR0);
}
```

As mentioned in the comments, you don’t have to worry too much about how the ISR works, just that it gets called whenever the microcontroller receives a character from the UART. Here, we check if the incoming character is the letter ‘a’, and we make sure that the OCR0 register is not at
its maximum value. If this is the case, we increment the value of the output compare register, which increases the duty cycle.

The overall period is determined by the size of our counter and the clock signal that we’re using. Since the ATmega16 has a default clock speed of 8MHz, and we’re using a prescale value of 256, the actual clock speed that the timer is using is:

\[ \frac{8 \text{ MHz}}{256} = 31.25 \text{ kHz} \]

which gives us a period of 32μs. When operating in Phase Correct PWM, one complete cycle consists of the counter counting all the way to 255 and back to zero, so one period of our output PWM signal is:

\[ 32 \mu s \times 256 \times 2 = 16.364 \text{ ms} \]

which is within our required range of 10ms – 20ms as required by the servo.

To set the duty cycle of our PWM signal, all we have to do is set the value of the OCR0 register, which will correspond to the length of the duty cycle. If OCR0 is equal to 0, we will have a 0% duty cycle. If OCR0 register is 255, we will have a 100% duty cycle.

Figure 7 below shows how OCR0 is used to control the duty cycle. When OCR0 is selected to be 50, the duty cycle will be 50/255*100% = 19.6%. When the frequency is sufficiently high, controlling the duty cycle of a PWM wave could actually be used in digital to analog conversion (i.e. DC motor speed control)

Figure 6. How OCR0 Controls Duty Cycle. The figure demonstrates how the OCR0 value is used to control the duty cycle for Phase Correct PWM. The OC0 pin is initially at 5V, and once the counter reaches the OCR0 value, the OC0 pin is set to low. OC0 is pulled high when the counter reaches the OCR0 value again.

[Go to the Atmel website and find the datasheet for the ATmega16. (Download the full datasheet, not the summary.) Find the section for TimeRCounter 0, and look over it. What bits of the TCCR0 register would need to be set if we wanted a clock prescale of 1024? What would the period of the output signal be? Hint: All of the bits that control the clock prescale are named CS0x.]

6. Now build your project and program the microcontroller. Check the fuses to ensure that your microcontroller’s clock is actually set to 8MHz. Now connect the serial cable to the RS232 Spare port and start HyperTerminal, and reset the STK500.
7. Press the ‘a’ key on your keyboard a couple of times. You should see the servo move and be able to observe the PWM signal on your oscilloscope. Add the “Period” and “Duty Cycle” measurements to your oscilloscope’s display. Continue to increase the duty cycle by pressing the ‘a’ key. Note that the period displayed on the oscilloscope may not exactly match the theoretical value of 16.364 ms. The ATmega16’s internal oscillator is not that precise, and any projects that require exact timing should use an external oscillator or external crystal. This discrepancy should have no effect on our servo’s performance though.

8. Continue to press the ‘a’ and ‘z’ keys on the keyboard. Find the values of OCR0 that correspond to the servo at the full left, full right, and center positions. The servos in lab only have 180° of travel so be sure that your servo is not actively trying to drive itself against one of the stops (maintain the OCR0 value within the range of 3-38). (You should be able to tell when this is happening by a noticeable twitch of the servo and hum of the motor. Leaving the servo in this state will only decrease its life.) Include these values, the theoretical “on” time of the duty cycle as displayed on HyperTerminal, and the actual “on” time of the duty cycle in your lab report.

4. Now create a new project and copy in the main_your_servo.c file from the ME 106 website. Use this template to write a program that makes the servo move all the way to the left when SW0 is pressed, all the way to the right when SW1 is pressed, and centered when no switches are pressed. Additionally, the corresponding LED should light up when the button is pressed. Be sure to include this code with your lab report. As always, your code should be properly formatted and indented to improve clarity and reduce the time you spend debugging.

Notes:
It is relatively easy to modify a servo so that it can rotate continuously. An excellent step-by-step guide for doing so can be found at: http://www.seattlerobotics.org/guide/servohack.html

If you use the Futaba FP-S148, it is especially easy. You don’t even need to replace the potentiometer as noted in the link above. You can manually position the pot shaft to its center position.

Once you’ve hacked the servo, you then have a low-cost, relatively powerful variable speed DC motor drive system that is great for any term projects or any other applications that needs a decent amount of accuracy.

Using a servo can be a low-cost and efficient approach for giving mobility to your robot. Tower Hobbies (http://www.towerhobbies.com/) has the best prices around for Futaba servos (3 for $39.47).
Appendix A – STK500 LED and Switch Implementation

Figure A1. Schematics for the switches and LEDs on the STK500 interface board. Output pins on the ATmega16 can sink enough current to turn on the LEDs, and thus can be connected directly to the LED’s header on the STK500. Likewise, the SWITCHES header pins can be connected directly to the pins of an input port on the ATmega16. Note that when a switch is pressed on the STK500 board, it connects the corresponding SWn pin to ground. Otherwise the SWn pin is held high by the 10kΩ pullup resistor. Also note that a LOW signal on the LEDn pin will cause the LED to turn on.