Parallax Feedback 360° High-Speed Servo (#900-00360)

The Parallax Feedback 360° High Speed Servo provides the functionality of a light-duty standard servo, continuous rotation servo, high-speed servo, and encoder in one convenient package.

Like most servos, the Feedback 360° is controlled by a 50 Hz PWM signal. What sets it apart is a return signal line from an internal Hall effect sensor system that provides digital angular position feedback.

Utilizing this feedback signal, your application can make the servo to turn to and hold any angle, much like a standard servo but with an unlimited range of motion. Or, rotate the servo continuously at a controlled, verifiable speed as a robot drive motor. Control signal response is nicely linear across the speed control range.

BlocklyProp blocks and user-friendly C Propeller C libraries are in the works. Low-level position control example C code is discussed in this guide.

Features

- Bidirectional, continuous, feedback-controllable, low-load rotation from -120 to 120 RPM
- PWM positional feedback across entire RPM range
- Internal Hall effect position sensor not subject to wear or sensor deadband as are potentiometer-style feedback systems
- No need to manually "center" the servo
- 3-pin ground-power-signal cable plugs onto the Activity Board's 3-pin header
- Separate single wire with female connector supplies feedback to an I/O pin

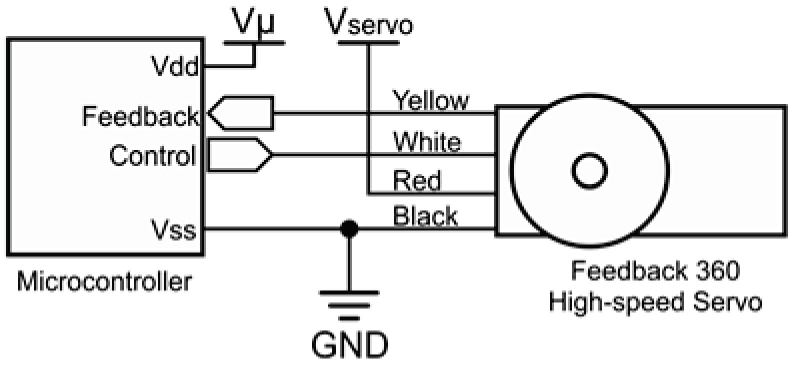
Specifications

- RPM: 120 w/feedback control, 140 max (+/- 15) @ 6 V, no load
- Gears: POM
- Case: Nylon & fiberglass
- Spline: 25-tooth, 5.96 mm OD
- Peak stall torque @ 6 V: 2.5 kg-cm (34.7 oz-in)
- Voltage requirements: 6 VDC typical, 5–8.4 VDC max range
- Current requirements: 15 mA (+/- 10) idle, 150 mA (+/- 40) no-load, 1200 mA stalled
- Control signal: PWM, 3–5 V 50 Hz, 1280–1720 µs
- Control signal zero-speed deadband: 1480–1520 µs (+/- 15)
- Feedback sensor: Hall effect
- Feedback signal: PWM, 3.3V, 910 Hz, 2.7%–97.1% duty cycle
- Product weight: 1.4 oz (40 g)
- Cable length: ~ 9.8 in (250 mm)
- Dimensions: approx. 2.15 x 1.46 x 0.79 in (50.4 x 37.2 x 20 mm)
- Mounting hole spacing: 10 x 49.5 mm on center
- Operating temperature range: 5 to 158 °F (-15 to +70 °C)

Resources and Downloads

Check for the latest version of this document, free software, and example programs from the Feedback 360° High-Speed Servo product page. Go to www.parallax.com and search 900-00360.

Quick-Start Circuit



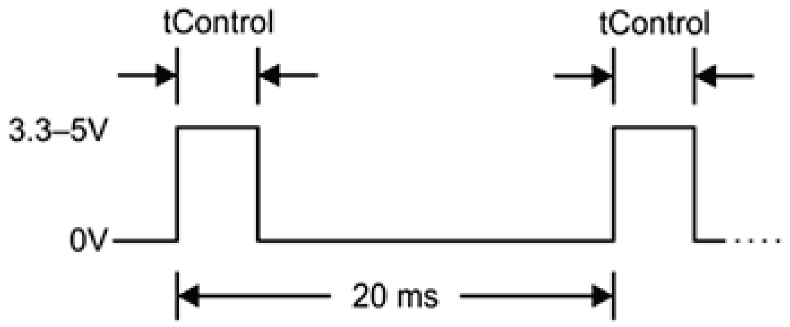
Pin Descriptions

Pin	Name	Description	Min	Typical	Max	Units
Yellow	Feedback	Servo output, PWM, 910 Hz, 2.7–97.1% duty cycle		3.3		V
White	Control	Servo input, PWM, 50 Hz, 1280–1720 µs	3.3	3.3	5.0	V
Red	Vservo	Servo power supply	5*	6	8.4	V
Black	Vss	Ground, common ground with microcontroller		0		v

*5 VDC is absolute minimum required for no-load angular position control. 5.8 to 8 VDC is recommended for continuous rotation speed control.

Control Signal

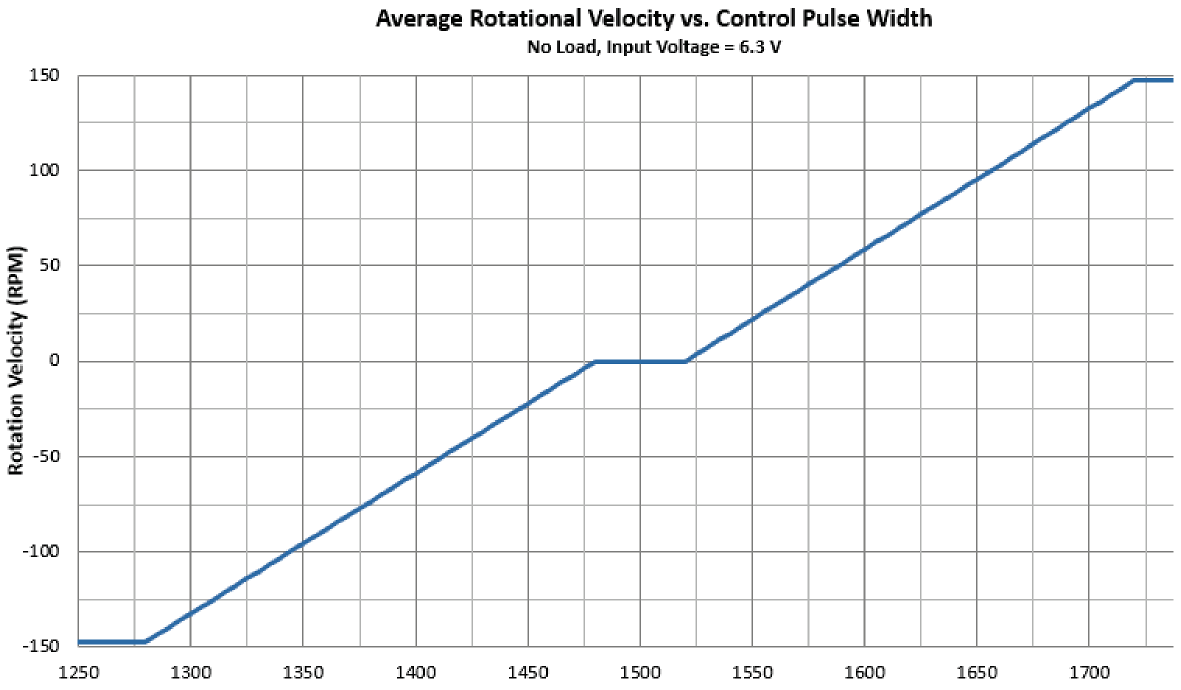
The servo requires an input control signal sent from your microcontroller via the white wire connection. This signal, labeled tControl in the diagram and table below, is the time in microseconds of a 3.3 to 5 V high pulse. The control signal must be sent every 20 ms.



The duration (pulse width) of tControl determines servo rotation speed and direction as shown in the table below. The top speed will vary with supply voltage and load.

	Clockwise (faster to slower)	Stop	Counterclockwise (slower to faster)
tControl, +/- 10 µs	1280….1480	1480...1520	1520….1720
RPM, +/-15	140...0	0	0…-140

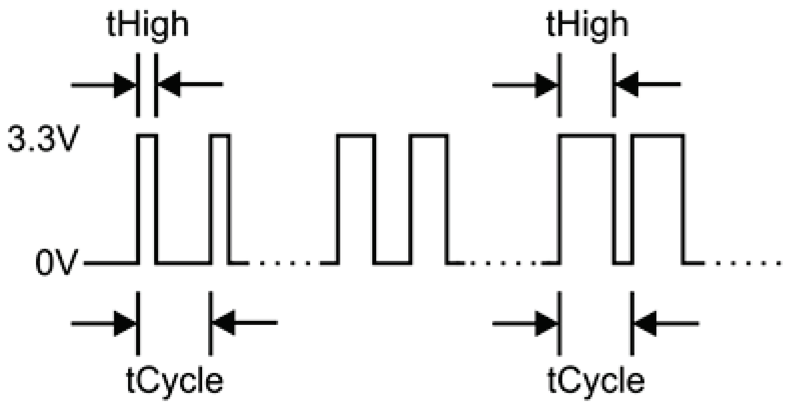
The rotational velocity response to tControl is fairly linear. This example graph is for 6.3 V, no load.



Feedback Signal

The servo sends a feedback output signal to your microcontroller via the yellow wire connection. This signal, labeled tCycle in the diagrams and equations below, has a period of 1/910 Hz (approx. 1.1 ms), +/- 5%.

Within each tCycle iteration, tHigh is the duration in microseconds of a 3.3 V high pulse. The duration of tHigh varies with the output of a Hall-effect sensor inside of the servo. The duty cycle of this signal, tHigh / tCycle, ranges from 2.9% at the origin to 91.7% approaching one clockwise revolution.



Duty cycle corresponds to the rotational position of the servo, in the units per full circle desired for your application.

For example, a rolling robot application may use 64 units, or "ticks" per full circle, mimicking the behavior or a 32-spoke wheel with an optical 1-bit binary encoder. A fixed-position application may use 360 units, to move and hold the servo to a certain angle.

The following formula can be used to correlate the feedback signal's duty cycle to angular position in your chosen units per full circle.

Duty Cycle = 100% x (tHigh / tCycle). Duty Cycle Min = 2.9%. Duty Cycle Max= 91.7%.

$$\textit{Angular position in units full circle} = \frac{\textit{(Duty Cycle - Duty Cycle Min) x units full circle}}{\textit{Duty Cycle Max - Duty Cycle Min + 1}}$$

The example code discussed in the Low-Level Programming section below demonstrates angular position control. It uses this formula along with a scale factor to convert floating-point percentages to integers. It also includes code to accommodate rotation that crosses the origin, and to compensate for starting positions other than the origin.

Programming Resources

C example code for low-level position control is available for download, and discussed in more detail in the next section. It is written for the Propeller microcontroller but should provide an illustration of programming strategies that can be adapted to other controllers.

Propeller C resources under development include two user-friendly libraries for higher-level control, servo360 and abdrive360. Both of these libraries support speed-controlled continuous rotation as well as angular position applications.

The abdrive360 library uses the same functions as the original abdrive library, so example programs that run on an ActivityBot robot using abdrive will also run using these Feedback 360° servos.

The servo360 library will be similar to the servo library, with enhancements to utilize the Feedback 360° servo's Hall-effect sensor output.

Blocks that use these libraries are also in the works for the BlocklyProp graphical programming system, to be released in Fall 2017.

Low-Level Position Control Programming

This section explains excerpts from this program: Feedback360 Angle Control [Low Level Example].side

This example Propeller C program uses two of the Propeller microcontroller's cogs (processors), one for angle detection and another for feedback control. It can all be done in the same loop, but it makes the code a little easier to digest in separate processors. These two lines launch the feedback and control functions into other cogs. Control here goes through the servo library, with valid speed values ranging from -220 (full speed clockwise) to -20...20 (stop) to 220 (full speed counterclockwise).

```
cog_run(feedback360, 128);
cog_run(control360, 128);
servo_speed(pinControl, -40);
```

This code allows you to enter angles into the terminal, and displays the servo's progress as it turns to its target position.

```
while(1)
{
  print("Enter angle: ");
  scan("%d", &targetAngle);
  print("\r");
  while(abs(targetAngle - angle) > 4)
  {
    print("targetAngle = %d, angle = %d\r", targetAngle, angle);
    pause(50);
  }
}
```

Position detection

The code from the `feedback360` function needs to be repeated rapidly, at least every ¼ turn, but between every servo pulse (50 Hz) is recommended.

This requires high and low pulse measurements for a duty cycle calculation. If your code skips a pulse between high and low, it also needs to check and make sure the cycle time is in the correct window. Reason being, if the high and low pulses are sampled before and after a moving servo crosses the 359 to 0 degree boundary, it might have a long high pulse for a high angle, and an almost equally long low pulse for a low angle, resulting in incorrect 180 degree calculation.

```
int tCycle = 0;
while(1)                                          // Repeat cycle time valid
{
  tHigh = pulse_in(pinFeedback, 1);               // Measure high pulse
  tLow = pulse_in(pinFeedback, 0);                // Measure low pulse
  tCycle = tHigh + tLow;                          // Calculate cycle time
  if((tCycle > 1000) && (tCycle < 1200)) break; // Cycle time valid?  Break!
}
dc = (dutyScale * tHigh) / tCycle;                // Calculate duty cycle
```

Next, calculate the angle `theta`. The `unitsFC` variable has a value like 360, for the number of units in a full circle The `dcMin` and `dcMax` values are 29 and 971. This calculation returns 0 to 359 depending on the measured duty cycle. Note that this uses (`unitsFC` - 1) - (the clockwise angle calculation). This gives us a positive counterclockwise angle to correspond with clockwise rotation being positive

```
theta = (unitsFC - 1) - ((dc - dcMin) * unitsFC) / (dcMax - dcMin + 1);
```

In case the pulse measurements are a little too large or small, let's clamp the angle in the valid range.

```
if(theta < 0) theta = 0;
else if(theta > (unitsFC - 1)) theta = unitsFC - 1;
```

Since we might be looking at positive or negative measurements above +/- 360 degrees, this keeps track of the number of full turns. `q2Min` is an abbreviation for quadrant 2 minimum, which is 90 degrees, and `q3Max` (quadrant 3 maximum) is 270 degrees. So, if the previous `theta` measurement was in the 1st quadrant, and the current `theta` measurement is in the 4th quadrant, we know we the angle transitioned from a high value to a low value, so increase the turns count. Conversely, if the previous theta was in the first quadrant, and the new theta is in the fourth quadrant, we know that the angle transitioned from a low to high value, so decrease the turns count.

```
if((theta < q2min) && (thetaP > q3max))          // If 4th to 1st quadrant
  Turns++;                                       // Increment turns count
else if((thetaP < q2min) && (theta > q3max))     // If in 1st to 4th quadrant
  turns --;                                      // Decrement turns count
```

This code takes the number of turns and the theta angle measurement, and constructs the total angle measurement from zero, allowing for large angle values in the +/- 2,147,483,647 degree range.

```
if(turns >= 0)
  angle = (turns * unitsFC) + theta;
else if(turns < 0)
  angle = ((turns + 1) * unitsFC) - (unitsFC - theta);
```

Since the 0 to 359 and 359 to 0 degree crossings rely on the angle from the previous time through the loop, the value of thetaP (theta previous) is copied from the current theta angle before the next loop repetition.

```
thetaP = theta;
```

Position Control System

Code from the control360 function is the bare minimum, using only proportional control to get the servo to turn to its target position. Proportional, integral, and derivative (PID) control with a position set point that marches forward at increments controlled by ramping and speed settings is one you might find inside abdrive360 or servo360.

Inside the loop, the first step is to check the difference between the angle set point and the measured angle.

```
errorAngle = targetAngle - angle;
```

For proportional control, this value can be multiplied by a constant. Kp is 1 in the example program.

```
output = errorAngle * Kp;
```

This code clamps the output to the max and min pulse speed values (assuming -speed is 1500 -20 to 1500 - 220 µs and + speed is 1500 + 20 to 1500 + 220 µs. .

```
if(output > 200) output = 200;
if(output < -200) output = -200;
```

This keeps the servo pushing slightly, even when it's close to zero degrees errorAngle. It also results in some oscillation. For best results, tune the 30 and -30 values for your servo.

```
if(errorAngle > 0)
  offset = 30;
else if(errorAngle < 0)
  offset = -30;
else
  offset = 0;
```

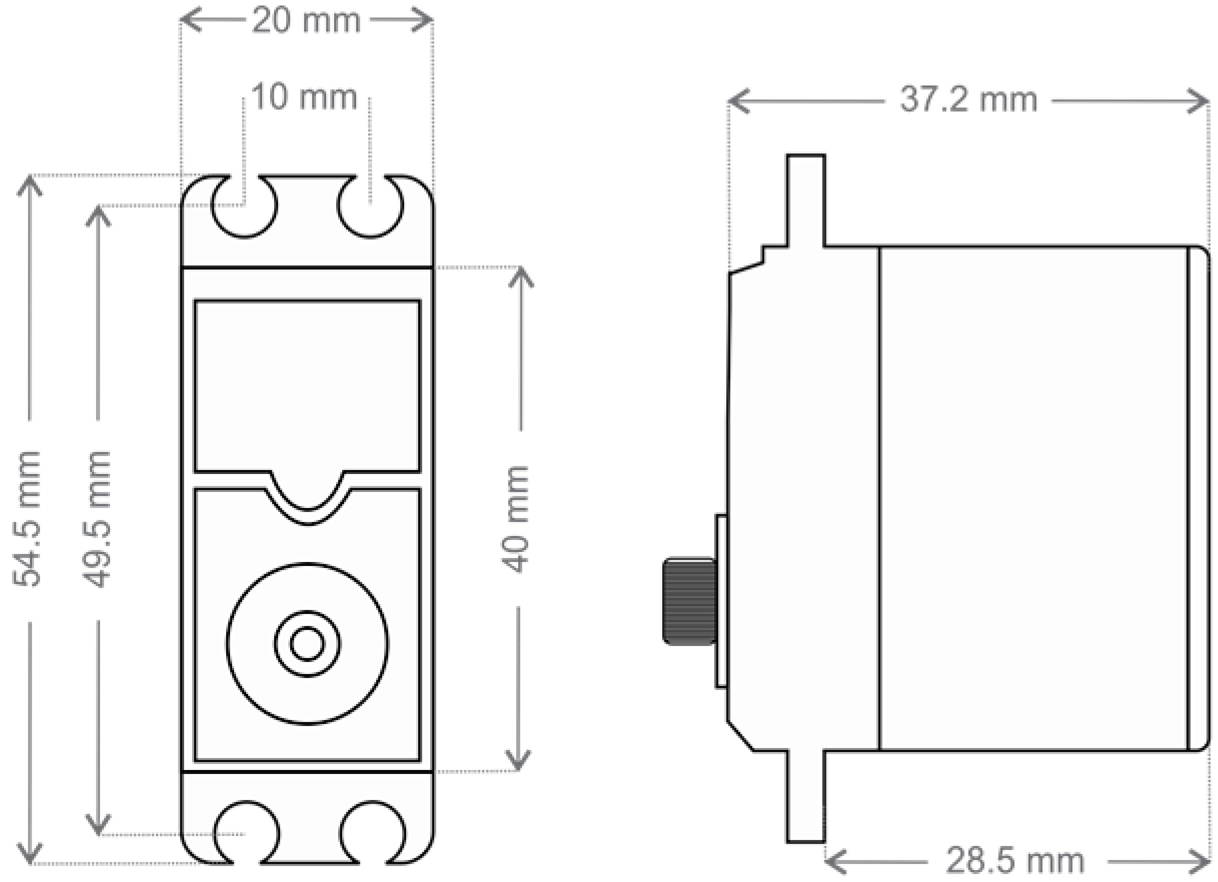
This uses the `output` and `offset` values to set the servo speed. When the error is large, the output will be large, up to 200. If there is any error, the offset will be 30 for + error or -30 for - error, or 0 for no error. So the resulting output will range from 30 to 230 for positive, counterclockwise correction, or -30 to -230 for negative, clockwise correction.

```
servo_speed(pinControl, output + offset);
```

After a 20 ms (1/50 s) delay, the loop repeats.

```
pause(20);
```

Module Dimensions



Revision History

Version 1.0: original release. Version 1.1: corrected 2 instances of 259 to 359 in example code discussion. Added statement below Pin Descriptions table.