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;
```

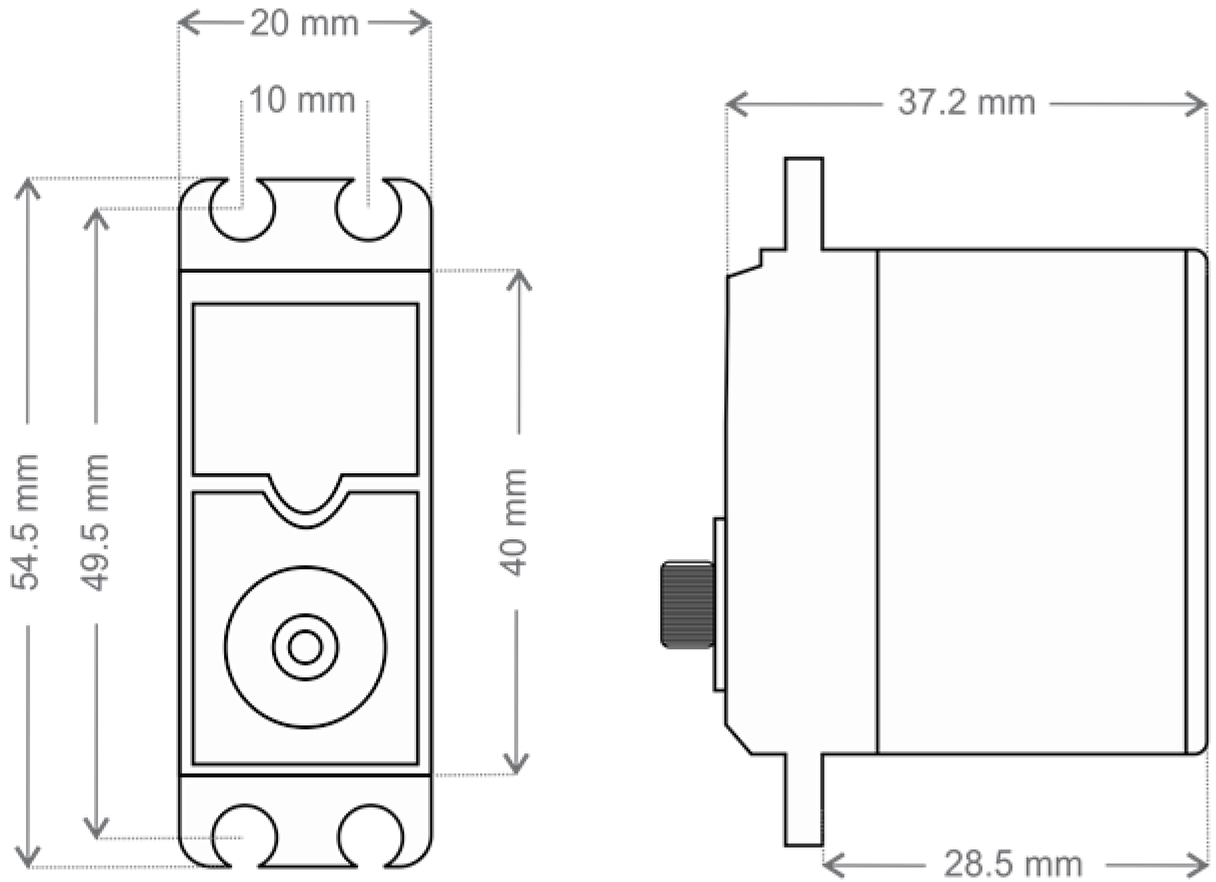
This uses the `output` and `offset` values to set the servo speed. When the error is large, the output will be large, up to 200. If there is any error, the offset will be 30 for + error or -30 for - error, or 0 for no error. So the resulting output will range from 30 to 230 for positive, counterclockwise 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. Version 1.2: corrected 2 instances of 91.7 to 97.1, pages 3 and 4.