Demystify SMC3Utils

PID Control Explained

A PID controller works by continuously comparing the desired position (**target**) to the actual position (**feedback**) and deciding how much motor effort is needed to reduce the difference between them. The **proportional (P)** term reacts immediately to the current error and provides the main driving force that moves the system toward the target; larger errors produce stronger output. The **integral (I)** term accumulates small errors over time and adds extra drive to eliminate steady offsets caused by friction or gravity, allowing the system to hold position accurately under load. The **derivative (D)** term looks at how quickly the position is changing and applies damping to slow the motion as it approaches the target, reducing overshoot and oscillation. Together, these three terms are summed to produce a single control output that is then limited, offset, and shaped before being sent to the motor driver. When tuned correctly, the PID loop causes the system to move quickly toward the target, settle smoothly without jitter, and remain stable even as loads and conditions change.

Kp (Proportional gain)

Description:

Kp (Proportional gain) controls how strongly the motor reacts to the current position error (difference between target and feedback). A higher Kp makes the actuator respond faster and feel stiffer, but too much causes overshoot and oscillation. Too little makes the system sluggish and unable to reach the target cleanly. Typical values are 200–300, with heavier rigs needing higher values. Kp is the primary "feel" knob and should be tuned first.

Typical values:

200 - 300

Technical:

Kp is the "push proportional to error" term: when the red Target line is away from the green Feedback line, Kp is the main reason the yellow output rises quickly to chase it. In your code it's used inside CalcMotor1PID() to build the P term from the instantaneous error.

```
pTerm_x100 = (Kp1_x100 + Kp1_x100) * (long)Error;     // Proportional
term
PWMout1 = (pTerm_x100 + iTerm_x100 + dTerm_x100) / 100;
```

If it does this → do that:
Fast wobble or buzzing around target → lower Kp.
Slow response → raise Kp.

Ki (Integral gain)

Description:
Ki (Integral gain) corrects steady-state error by accumulating small errors over time and pushing the motor until the error disappears. It helps the rig hold position against constant loads like gravity, but too much Ki causes slow oscillation or drift. Ki should be much smaller than Kp;

Typical values:
5–20.

Technical:
Ki accumulates error over time to eliminate steady "can't quite reach target" behavior.

If it does this → do that:
Holds off-target under load → increase Ki slightly.
Slow swaying or drift → reduce Ki.

Kd (Derivative gain)

Description:
Kd (Derivative gain) dampens motion by reacting to how fast the error is changing. It reduces overshoot and ringing, especially on fast or light mechanisms. Too much Kd makes the system feel "sticky" or noisy. Kd is most useful once Kp is already close to optimal.

Typical values:
Set to 5–15.

Technical:
Kd damps motion by reacting to how quickly the system is changing; it helps prevent overshoot when yellow is high and green is moving fast toward red. In the code the D term is based on DeltaPosition. Too much Kd makes motion feel sticky/noisy; too little makes it ring.

```
dTerm_x100 = (Kd1_x100 * (long)DeltaPosition);       // Derivative term
DeltaPosition = CurrentPosition - PrevPosition;
```

If it does this → do that:
Overshoot/ringing after a move → increase Kd
A bit; sluggish "draggy" response → reduce Kd.

Ks (PID speed scaling)

Description:
Ks (PID speed scaling) adjusts how aggressively the PID output is scaled relative to motion speed. In most SMC3 builds this is left at 1, and changing it is rarely necessary

Typical values:
For most rigs, Ks = 1

Technical:
Ks scales the final PID output before it is applied to the motor command. This value is effectively unity and not normally altered.

If it does this → do that:
Unexpected gain changes with speed → return Ks to 1.

PWMmin

Description:
PWMmin defines the minimum motor command that will ever be applied once the motor is allowed to move. This compensates for motor deadband, friction, and gearbox friction. If PWMmin is too low, the motor won't start moving smoothly; if it's too high, the rig will feel jerky near the target.

Typical values:
20–30 for Sabertooth-driven DC motors.

Technical:
PWMmin is the minimum shove you add whenever you decide to move (outside deadzone). It mainly affects yellow near the target: with a high PWMmin, yellow won't taper smoothly; it "snaps" into motion. In your code, PWMmin is literally added before sending the Sabertooth command.

```
PWMout1 += PWMoffset1;                                  // Add minimum
drive (PWMmin)
if(PWMout1 > PWMmax1){ PWMout1 = PWMmax1; }             // Clamp after
offset
```

If it does this → do that:
Motor won't start moving until error grows → increase PWMmin;
Jerky near target → decrease PWMmin.

PWMmax

Description:
PWMmax limits the maximum motor command sent to the driver. It caps speed and protects hardware during tuning. Higher values give faster motion but increase stress and instability. A safe tuning range is 80–120, increasing only after the system is stable.

Technical:
PWMmax caps the maximum effort (limits how tall the yellow line can get during big moves). It's your "don't go full send yet" safety limiter during tuning. In your code, PWMout is clamped before converting to Sabertooth scale.

```
if(PWMout1 > PWMmax1){ PWMout1 = PWMmax1; }           // Max clamp
(normal)
ST[0].motor(1, constrain((PWMout1/2), -127, 127));     // Send to
Sabertooth
```

If it does this → do that:
Too slow overall / can't keep up → increase PWMmax;
Too violent / slams → decrease PWMmax.

PWMrev

Description:
PWMrev is a fixed "push-back" power used only when feedback enters the input clipping zone. It bypasses PID and forces the actuator away from the end of travel. This is not proportional control; it is a safety shove.. Values that are too high cause banging and oscillation at the limits.

Typical values:
40–80

Technical:
PWMrev is a fixed "get out of the end zone" command used only when you enter the clip region. On the plot, this often looks like yellow snapping to a flat level near the end even if PID would do something else. In your code it overrides PID completely when feedback crosses InputClipMin/Max.

```
if((Feedback1 > InputClipMax1) && (PWMrev1 != 0)) { PWMout1 = PWMrev1;
}
if((Feedback1 < InputClipMin1) && (PWMrev1 != 0)) { PWMout1 = PWMrev1;
}
```

If it does this → do that:
Chattering/square-wave yellow near clip boundary → lower PWMrev
It fails to back off at the ends → increase PWMrev.

Fpwm

Description:
Fpwm sets the PWM frequency used by the older H-Bridge motor driver. Energy360.ino only supports Sabertooth in packet serial mode, this value changes nothing. In SMC3Utils and typically left at 25 kHz.

Technical:
Fpwm does not Motor drives in Energy360.ino

If it does this → do that:
leave unchanged unless using a direct PWM H-bridge.

Deadzone

Description:
Deadzone defines a tolerance band around the target position where the controller stops driving the motor and commands zero output. It prevents jitter, buzzing, and constant correction due to ADC noise and mechanical play. Typical values are 10–15 ADC counts on an Arduino Uno. Deadzone must be tuned together with PWMmin.

Typical values:
10–15

Technical:
Deadzone is the "close enough → stop driving" band. When green Feedback is within ±Deadzone of red Target, your output function hits the brake branch and commands yellow to zero (Sabertooth stop). If Deadzone is 0, you almost never settle because ADC noise keeps the error nonzero, and the motor keeps turning

```
else { ST[0].motor(1, 0); }                              // Brake/stop
inside deadzone
if ((Target1 > (Feedback1 + DeadZone1)) || (Target1 < (Feedback1 -
DeadZone1))) { ... }
```

If it does this → do that:
Constant twitching near target → raise Deadzone (e.g., 10–15);
Feels sloppy/inaccurate → lower Deadzone.

Clip Input

Description:
Clip Input defines a soft-limit region near the ends of travel where normal PID control is overridden and PWMrev is applied to push the actuator back into the safe operating range. This region should be comfortably inside the hard cutoff limits. Typical values are 50–150, depending on sensor resolution and mechanical stroke.

Typical values:
50-150

Technical:
Clip Input defines a soft boundary near travel extremes where you stop trusting PID and start applying PWMrev "push-back." On the plot this is where yellow behavior changes mode near the ends, usually before the hard limit lines. If the clip region is too wide, you spend too much time in push-back logic; too narrow and you'll hit hard limits more often.

```
if(Feedback1 > InputClipMax1) { /* clip behavior */ } // Upper soft
clip
if(Feedback1 < InputClipMin1) { /* clip behavior */ } // Lower soft
clip
```

If it does this → do that:
You constantly hit push-back during normal travel → reduce clip band;
You hit hard limits → move clip inward so it triggers sooner.

Max Limits

Description:
Max Limits define the maximum target range that SMC3Utils will command. These should match the safe physical travel of the actuator so the software never asks for unreachable positions. Typical values are 20–30, adjusted so the rig never reaches hard cutoff during normal motion.

Typical values:
20–30

Technical:
These are the hard safety fences shown as red limit lines: cross them and the code disables the motor. This was not working in original code and Energy360.ino fixes this and commands the Sabertooth to stop immediately. On the plot you'll see green hit the limit line and yellow collapse to zero/stop. If you're hitting these during normal operation, your target range is too big or your clip is too late.

```
if ((Feedback1 > CutoffLimitMax1) || (Feedback1 < CutoffLimitMin1)) {
DisableMotor1(); }
ST[0].motor(1, 0);                                    // Keep stopped
while disabled
```

If it does this → do that:
Hits hard limit → reduce Max Limits in SMC3 or increase Clip action earlier; never reaches full stroke → increase Max Limits carefully.
