

PIDControl for Arduino

PID control, process identification, tuning and filtering

PIDControl v1.4.2 Documentation

Scope. This document describes the PIDControl v1.4.2 library API and, in separate chapters, the TCLab sketch used for hardware validation. Console commands belong to the test sketch; the core library does not depend on Serial.

19/09/2026

1. Introduction: what PIDControl provides	4
2. Library architecture.....	5
What is new in version 1.4.2.....	5
3. Terminology, variables and units	6
4. Controller construction and basic cycle	7
5. PID, PI-D and I-PD structures	8
6. DIRECT and REVERSE action	9
7. MAN/AUTO modes, PV Tracking and bumpless transfer.....	10
MAN mode	10
AUTO mode	10
PV Tracking	10
Bumpless transfer	10
8. Saturation, anti-windup, limits and sample time	11
Output limits	11
Anti-windup	11
Sample time	11
9. Signal conditioning and filters	12
Moving average	12
Median	12
First-order low-pass	12
Complementary filter.....	12
10. Open-loop identification with StepTest	13
Work sequence	13
Robust steady-state criterion.....	13
Directional response	13
11. FOPDT model and normalized gain.....	14
Gain in engineering units.....	14
Estimation of T0 and Tp	14
Normalized gain.....	14
FOPDTModel object	14
12. IMC/Lambda and Tyreus-Luyben tuning with PIDTuning.....	15
IMC / Lambda PI.....	15
Lambda PID	15
13. Relay experiment with RelayTest	16
Compact API configuration.....	16
Operation	16
Calculation of Ku and Tu	16

Result persistence in memory.....	17
Real TCLab validation example.....	17
Tyreus-Luyben integrated in PIDControl 1.4.2.....	17
14. Test states, errors and safe behavior	17
StepTestState	17
StepTestError	18
RelayTestState	18
RelayTestError.....	18
15. TCLab sketch: acquisition, filtering, safety and validation	20
Acquisition chain.....	20
Operating range and thermal protection.....	20
16. TCLab sketch serial-command interface	21
General commands	21
StepTest commands.....	21
RelayTest commands.....	22
Compact RelayTest format.....	22
Retrieving results with IDENT.....	23
Frame for Arduino COM Plotter.....	23
17. Recommended workflows	24
Normal operation	24
FOPDT identification + IMC/Lambda.....	24
Relay experiment.....	24
18. Complete API reference	25
PIDControl	25
StepTest.....	25
RelayTest.....	26
PIDTuning.....	26
PIDControlFilters	27
19. Limitations and good practices.....	28
20. Package structure and repository preparation.....	29
Installation.....	29
Appendix A. Complete minimal example	30
Appendix B. Quick reference	31

1. Introduction: what PIDControl provides

PIDControl is an Arduino library aimed at process control. Its purpose is not limited to calculating a PID output, but to cover a complete chain: signal conditioning, MAN/AUTO control, process identification, tuning and validation.

The library follows a philosophy close to industrial instrumentation: it separates PV, SP and OP; distinguishes direct and reverse action; implements PV tracking, bumpless transfer, anti-windup and several controller structures. Identification tests are non-blocking and work with already scaled variables, so the hardware remains outside the algorithm.

- PID controller with PID, PI-D and I-PD structures.
- MAN and AUTO modes.
- DIRECT and REVERSE action.
- PV Tracking and bumpless transfer.
- Conditional-integration anti-windup.
- Configurable output limits and sample time.
- Moving-average, median, low-pass and complementary filters.
- StepTest for open-loop FOPDT identification.
- Process gain in engineering units and normalized gain.
- PIDTuning for IMC/Lambda PI, Lambda PID and Tyreus-Luyben tuning.
- RelayTest to obtain K_u and T_u using a relay experiment.

Main idea. PIDControl aims to keep application code limited to reading PV, calling the control/identification classes and writing OP. Control logic should not be mixed with pins, ADC, PWM or communications.

2. Library architecture

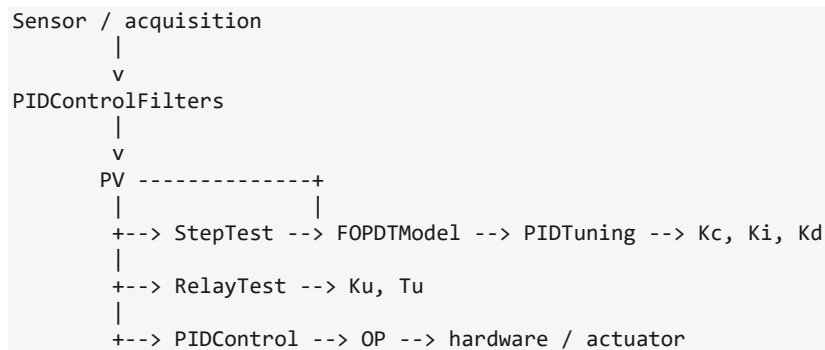
PIDControl v1.4.2 is divided into five main blocks. This separation allows each project to use only the modules it needs.

What is new in version 1.4.2

Version 1.4.2 completes the identification and tuning chain. StepTest provides an IMC/Lambda PI tuning and a Lambda PID tuning, while RelayTest can transform Ku/Tu into Tyreus-Luyben PI and PID proposals. The TCLab sketch adds the compact RELAY(...) command and IDENT to retrieve results stored in memory.

Module	Responsibility	Header
PIDControl	Closed-loop control: PID, PI-D, I-PD, MAN/AUTO, action, tracking, bumpless transfer and anti-windup.	PIDControl.h
StepTest	Non-blocking step-response experiment and FOPDT identification.	StepTest.h
RelayTest	Non-blocking relay experiment to estimate Ku and Tu.	RelayTest.h
PIDTuning	IMC/Lambda PI and Lambda PID tuning from FOPDT; Tyreus-Luyben PI/PID from Ku/Tu.	PIDTuning.h
PIDControlFilters	Reusable signal filters.	PIDControlFilters.h

The typical relationship between the modules can be represented as follows:



StepTest and RelayTest receive pointers to PV and OP. They do not know which pin is used or how a reading is scaled. This allows the same identification logic to be used with ADC/PWM, I2C, SPI, Modbus, a simulator or any other hardware layer.

3. Terminology, variables and units

Symbol	Meaning	Example
PV	Process Variable. Measured process variable.	Temperature T1 [°C]
SP	Setpoint. Desired target value.	50 °C
OP	Output. Controller output.	0..100 %
Kc	Controller proportional gain.	6.40
Ki	Integral gain used internally [1/s].	0.0426 1/s
Kd	Derivative gain used internally [s].	0
Ti	Integral time.	150 s
Td	Derivative time.	0 s
Kp	Process gain: $\Delta PV / \Delta OP$.	°C/%OP
T0	Process dead time.	s
Tp	Process time constant.	s
Ku	Ultimate gain estimated by relay experiment.	dimensionless depending on scaling
Tu	Ultimate period.	s

Tuning-parameter conversion

$$K_i = K_c / T_i$$

$$K_d = K_c * T_d$$

PIDControl uses seconds in the controller equations. Therefore Ti, Td and the time basis associated with Ki/Kd must be consistent with seconds.

Kc is not Kp. In this documentation Kc is reserved for controller gain and Kp for the identified process gain. This distinction avoids a common ambiguity in PID documentation.

4. Controller construction and basic cycle

The constructor receives pointers to PV, OP and SP together with Kc, Ki and Kd. The shortest declaration uses the library defaults and is usually sufficient for a typical configuration.

```
PIDControl pid(&PV, &OP, &SP, Kc, Ki, Kd);

// Full declaration
const bool pvTracking = true;

PIDControl fullPid(
    &PV, &OP, &SP,
    Kc, Ki, Kd,
    PIDType::PI_D,
    PIDAction::REVERSE,
    pvTracking
);
```

With the short declaration the controller starts in MAN mode, with PI_D structure, REVERSE action, PV Tracking enabled, output limits from 0 to 100 % and a sample time of 100 ms. The full declaration makes these parameters explicit or allows them to be replaced by other values.

```
void loop()
{
    PV = ReadProcessVariable();

    pid.Compute();

    WriteOutput(OP);
}
```

Compute() returns true when a new automatic calculation has been performed. In AUTO it respects the configured sample time. In MAN it applies the manual output and, if PV Tracking is enabled, updates the internal SP so that it follows the PV.

Important. Compute() should also be called in MAN. In PIDControl, manual mode does not mean the class stops participating: it maintains manual OP and, when applicable, PV tracking.

5. PID, PI-D and I-PD structures

PIDControl supports three controller structures through PIDType. The default structure is PI_D.

Structure	P acts on	I acts on	D acts on	Conceptual use
PID	Error	Error	Error	Classic PID.
PI_D	Error	Error	PV	Avoids derivative kick on SP changes.
I_PD	PV	Error	PV	Reduces the immediate effect of SP changes on P and D.

For REVERSE action, with $e = SP - PV$, the implemented equations are:

```

PID:
P = Kc * e
I(k) = I(k-1) + Ki * e * Ts
D = Kd * (e - previousError) / Ts

PI-D:
P = Kc * e
I(k) = I(k-1) + Ki * e * Ts
D = -Kd * (PV - previousPV) / Ts

I-PD:
P = -Kc * PV
I(k) = I(k-1) + Ki * e * Ts
D = -Kd * (PV - previousPV) / Ts

```

The structure can be selected at any time with:

```

pid.SetStructure(PIDType::PID);
pid.SetStructure(PIDType::PI_D);
pid.SetStructure(PIDType::I_PD);

```

If the structure is changed while the controller is in AUTO, PIDControl recalculates the integral term through InitializeBumpless() to minimize the OP step.

6. DIRECT and REVERSE action

Before putting a controller into automatic mode, we must verify that the control action is correct. The simplest way to choose it is to ask what OP should do when PV increases. If OP must decrease when PV increases, use REVERSE. If OP must also increase when PV increases, use DIRECT. PIDControl uses REVERSE by default.

Action	Typical behavior	Example
REVERSE	If PV increases, OP decreases.	Heating: if temperature increases, power is reduced.
DIRECT	If PV increases, OP increases.	Processes in which PV and OP must change in the same direction.

```
pid.SetDirection(PIDAction::REVERSE);
pid.SetDirection(PIDAction::DIRECT);
```

TCLab. In the thermal process used in the examples, if temperature increases we want to reduce heater power. Therefore REVERSE action is used.

7. MAN/AUTO modes, PV Tracking and bumpless transfer

MAN mode

In MAN, the output is set with SetManualOutput().

```
pid.SetMode(PIDMode::MAN);
pid.SetManualOutput(40.0);
```

AUTO mode

In AUTO, the output is calculated by the controller.

```
pid.SetMode(PIDMode::AUTO);
```

PV Tracking

PV Tracking is enabled by default. While the controller remains in MAN, the internal SP follows the PV and the operator-requested setpoint is stored separately in operatorSetpoint. This makes the MAN-to-AUTO transition easier and prepares a transfer without an abrupt output step.

```
pid.SetPVTracking(true);
pid.SetSetpoint(50.0);    // stored as the operator setpoint

// In MAN + tracking, the active SP follows PV.
// The 50.0 setpoint remains available through GetOperatorSetpoint().
```

Bumpless transfer

When switching to AUTO, changing Kc/Ki/Kd, modifying the structure, reversing the action or changing output limits while the controller is in AUTO, PIDControl initializes the integral so that $P + I + D$ starts approximately from the current OP.

Conceptual initialization

```
I = current_OP - P
```

The derivative contribution is initialized to zero because previousPV and previousError are updated with the current values.

8. Saturation, anti-windup, limits and sample time

Output limits

```
pid.SetOutputLimits(0.0, 100.0);
```

The calculated OP and the manual OP are limited to the configured interval. The default limits are 0..100 %. If the limits are changed in AUTO, a new bumpless initialization is performed.

Anti-windup

PIDControl uses conditional integration. Before accepting the new integral value it calculates an unsaturated OP. If the output is above the maximum and the integral would increase it even further, integration is stopped. The same logic applies at the lower limit.

```
integralIncrement = sign * Ki * error * Ts;
newIntegral       = integral + integralIncrement;
unsaturatedOutput = P + newIntegral + D;

if unsaturated OP > max and integralIncrement > 0 -> do not integrate
if unsaturated OP < min and integralIncrement < 0 -> do not integrate
```

Sample time

```
pid.SetSampleTime(100); // ms
```

The default sample time is 100 ms. Compute() performs a new automatic calculation only after at least the configured period has elapsed. The equation uses the actual elapsed time converted to seconds rather than assuming a fixed Ts constant.

9. Signal conditioning and filters

PIDControlFilters is independent of the controller and can process the signal before it is passed to the PID or to the identification tests. The goal is not to obtain the prettiest possible curve, but to remove enough noise without hiding the real process dynamics.

Moving average

```
MovingAverageFilter<10> filter;
double filteredPV = filter.Update(rawPV);
```

Equation

```
y[k] = (1/N) * sum(x[k-i]), i=0..N-1
```

It reduces random noise at the cost of adding delay. During startup it averages only the samples actually received until N samples are available.

Median

```
MedianFilter<5> filter;
double filteredPV = filter.Update(rawPV);
```

It is especially effective against isolated spikes and outliers. The implementation supports both even and odd window sizes.

First-order low-pass

```
LowPassFilter filter(1.0); // tau = 1 s
double filteredPV = filter.Update(rawPV, dtSeconds);
```

Exact discrete coefficient

```
alpha = 1 - exp(-dt/tau)
y[k] = y[k-1] + alpha * (x[k] - y[k-1])
```

The public parameter is the physical time constant tau, not an arbitrary smoothing factor.

Complementary filter

```
ComplementaryFilter filter(0.98);

double fused = filter.Update(fastEstimate, slowReference);
double angle = filter.UpdateIntegrated(angularRate, absoluteAngle, dt);
```

The complementary filter needs two estimates of the same quantity with complementary properties. It should not be interpreted as a simple low-pass filter for a single signal.

Filtering and identification. Every filter adds dynamics. Excessive filtering can make the process appear to have a larger dead time or time constant during identification than it really has. Filter only as much as necessary.

10. Open-loop identification with StepTest

StepTest runs a step-response experiment without blocking the Arduino loop. Its purpose is to estimate a FOPDT model from one or more OP changes.

```
StepTest test(pid, &PV, &OP);

test.SetStepSize(10.0);
test.SetNumberOfSteps(2);
test.SetFirstStepUp(true);
test.SetProcessGain(ProcessGain::POSITIVE);

test.SetStabilityCriteria(0.30, 60.0, 0.10);
test.SetResponseThreshold(0.40);
test.SetMaxTestTime(30.0);

test.Start();
```

During the loop:

```
test.Update();
pid.Compute();
```

Work sequence

- Forces the controller to MAN.
- Stores the current OP as the bias.
- Waits for an initial steady state.
- Applies the configured steps alternately.
- Checks that PV responds in the expected direction.
- Waits for the new steady state.
- Calculates a model for each valid step.
- Averages the valid models.
- When finished or aborted, restores OP to the original bias and leaves the PID in MAN.

Robust steady-state criterion

Simultaneous conditions

```
PVmax - PVmin <= 2 * band
abs(regression_slope) <= maxSlope
```

The slope is obtained by least-squares linear regression over the stability window and is expressed in PV units per minute.

Directional response

ProcessGain::POSITIVE means that an increase in OP must eventually increase PV. NEGATIVE means the opposite. This criterion prevents residual inertia in the wrong direction from being mistaken for the step response.

11. FOPDT model and normalized gain

StepTest estimates a first-order plus dead-time model:

Model

$$G(s) = K_p * \exp(-T_0*s) / (T_p*s + 1)$$

Gain in engineering units

Process gain

$$K_p = \Delta PV / \Delta OP$$

Estimation of T0 and Tp

The implementation uses the 28.3 % and 63.2 % response crossings:

28.3/63.2 method

$$\begin{aligned} T_p &= 1.5 * (t_{63.2} - t_{28.3}) \\ T_0 &= 1.5 * t_{28.3} - 0.5 * t_{63.2} \end{aligned}$$

Normalized gain

The test safety limits and the full engineering ranges are different concepts. The ranges are used only to normalize the gain.

```
test.SetOutputLimits(0.0, 60.0); // test safety limit
test.SetPVLimits(15.0, 45.0);   // test safety limit

test.SetOutputRange(0.0, 100.0); // full OP range
test.SetPVRange(0.0, 80.0);     // full PV range
```

Normalized gain

$$\begin{aligned} K_{p\text{Normalized}} &= (\Delta PV / PV_{\text{span}}) / (\Delta OP / OP_{\text{span}}) \\ &= K_p * OP_{\text{span}} / PV_{\text{span}} \end{aligned}$$

KpNormalized is expressed as %PV/%OP and is reported only when both engineering ranges are configured.

FOPDTModel object

```
FOPDTModel model = test.GetModel();

model.valid
model.Kp
model.KpNormalizedValid
model.KpNormalized
model.T0
model.Tp
model.pvRangeMin
model.pvRangeMax
model.outputRangeMin
model.outputRangeMax
```

This object decouples identification from tuning and can be passed directly to PIDTuning.

The StepTest result also remains in the object after the test has finished. In the TCLab v1.4.2 sketch it can be retrieved with IDENT STEP as long as RESET is not executed, the board is not restarted and another StepTest does not overwrite the state.

12. IMC/Lambda and Tyreus-Luyben tuning with PIDTuning

PIDTuning converts identification results into controller parameters without applying them automatically. From a FOPDT model it provides an IMC/Lambda PI tuning and a Lambda PID tuning. From Ku and Tu obtained by RelayTest it provides Tyreus-Luyben PI and PID tunings.

```
PIDTuning tuning(model);

PIDTuningResult pi = tuning.IMC_PI();
PIDTuningResult pid = tuning.Lambda_PID();

PIDTuningResult piRobust = tuning.IMC_PI(IMCSpeed::ROBUST);
PIDTuningResult pidRobust = tuning.Lambda_PID(IMCSpeed::ROBUST);
```

IMC / Lambda PI

```
Kc = Tp / (abs(Kp) * (Tf + T0))
Ti = Tp
Td = 0
Ki = Kc / Ti
Kd = 0
```

Preset	Tf
IMCSpeed::AGGRESSIVE	$Tf = 1 \cdot T0$
IMCSpeed::NORMAL	$Tf = 2 \cdot T0$
IMCSpeed::ROBUST	$Tf = 3 \cdot T0$

Overloads without parameters use the NORMAL preset, so $Tf = 2 \cdot T0$ for both PI and PID. If $T0$ is numerically zero, PIDControl uses $0.1 \cdot Tp$ as a fallback value. AGGRESSIVE, NORMAL and ROBUST are practical library presets, not universal rules of the method.

Lambda PID

The Rivera/Lambda PID rule uses the same FOPDT model and a Tf parameter that sets the desired closed-loop response speed:

```
Kc = (Tp + T0/2) / (abs(Kp) * (Tf + T0/2))
Ti = Tp + T0/2
Td = Tp*T0 / (2*Tp + T0)
Ki = Kc / Ti
Kd = Kc * Td
```

In PIDControl, `Lambda_PID()` returns Kc , Ti , Td , Ki and Kd . The ``lambda`` field in `PIDTuningResult` keeps that name for API compatibility, but in this documentation it represents the Tf value used to calculate the tuning.

```
PIDTuningResult pid = tuning.Lambda_PID();
PIDTuningResult pid60 = tuning.Lambda_PID(60.0);
PIDTuningResult pidRobust = tuning.Lambda_PID(IMCSpeed::ROBUST);
```

Sign of Kp . PIDTuning uses `abs(Kp)` to calculate a positive Kc . Controller direction is handled independently through `PIDAction`.

Applying the tuning is explicit:

```
PIDTuning::Apply(pid, pi);
```

`Apply()` changes only Kc , Ki and Kd . It does not modify structure, mode, action or SP.

13. Relay experiment with RelayTest

RelayTest implements a non-blocking experiment inspired by the Åström-Hägglund relay autotuning principle. Its role is to identify K_u and T_u ; the conversion of those values into controller parameters is performed afterwards by PIDTuning.

```
RelayTest relay(pid, &PV, &OP);

relay.SetRelayAmplitude(10.0);
relay.SetHysteresis(0.5);
relay.SetCycles(3);
relay.UseCurrentPVAsReference();
relay.SetMaxTestTime(15.0);
relay.SetOutputLimits(20.0, 60.0);
relay.SetPVLimits(15.0, 48.0);

relay.Start();
```

Compact API configuration

Since v1.4.1, RelayTest itself provides RelayTestConfig and Start(config):

```
RelayTestConfig cfg;
cfg.amplitude = 10.0;
cfg.hysteresis = 0.5;
cfg.cycles = 3;
cfg.useCurrentReference = true;
cfg.maxTestTimeMinutes = 15.0;
cfg.outputMin = 20.0;
cfg.outputMax = 60.0;
cfg.pvMin = 15.0;
cfg.pvMax = 48.0;

relay.Start(cfg);
```

Important distinction. RelayTestConfig is part of the library. The textual RELAY(...) syntax is the responsibility of a specific sketch parser; PIDControl does not directly interpret Serial commands.

Operation

Relay levels

```
OPhigh = bias + d
OPlow = bias - d
```

The bias is the OP that exists when the test starts. By default, the relay reference is the PV present at Start(). For REVERSE action, normally associated with positive process gain, the logic is:

```
PV >= reference + hysteresis -> OP low
PV <= reference - hysteresis -> OP high
```

With DIRECT action, the switching direction is reversed.

Calculation of K_u and T_u

Ideal symmetric-relay approximation

```
a = (PVmax - PVmin) / 2
Ku = 4*d / (pi*a)
Tu = average period between equivalent transitions
```

The first complete oscillation is discarded as a startup transient. The number configured with SetCycles() corresponds to the subsequent cycles used in the averages.

Result persistence in memory

GetResult() can be queried again after the experiment has finished. In the TCLab v1.4.2 sketch, IDENT RELAY prints Ku, Tu and the Tyreus-Luyben proposals again while the result remains stored in memory and has not been reset or overwritten by another relay experiment.

Integrated tuning in v1.4.2. RelayTest remains limited to identifying Ku and Tu. PIDTuning adds Tyreus-Luyben PI and PID as a separate stage. The library never applies a tuning automatically: the user decides which one to use.

Real TCLab validation example

In the TCLab validation, the experiment was run from an OP bias of 40 %, with a relay amplitude of 10 %, 0.5 °C hysteresis and three valid cycles. OP oscillated between 30 % and 50 %, and the test finished by restoring the original bias in MAN.

```
Ku = 14.233237
Tu = 180.300 s
PV amplitude = 0.903066 °C
Relay amplitude = 10 %
Valid cycles = 3
```

As an external check, these values can be used to calculate a Ziegler-Nichols tuning. For PI, using $K_c = 0.45 \cdot K_u$ and $T_i = T_u/1.2$, the approximate result is:

```
Kc = 6.404957
Ti = 150.250 s
Ki = Kc/Ti = 0.042629 1/s
Kd = 0
```

Tyreus-Luyben integrated in PIDControl 1.4.2

From the same Ku and Tu, PIDTuning directly provides two alternatives that are less aggressive than classic Ziegler-Nichols:

```
Tyreus-Luyben PI:
Kc = Ku / 3.2
Ti = 2.2*Tu
Ki = Kc/Ti

Tyreus-Luyben PID:
Kc = Ku / 3.3
Ti = 2.2*Tu
Td = Tu / 6.3
Ki = Kc/Ti
Kd = Kc*Td
```

With the real experiment $K_u=14.233237$ and $T_u=180.300$ s, the approximate result is:

```
Tyreus-Luyben PI: Kc=4.447887 Ti=396.660 s Ki=0.011213 Kd=0
Tyreus-Luyben PID: Kc=4.313102 Ti=396.660 s Td=28.619 s Ki=0.010874 Kd=123.437
```

Application remains explicit. PIDTuning::Apply() or SetTunings() modify the tuning only when the user requests it.

14. Test states, errors and safe behavior

StepTestState

State	Meaning
IDLE	No active test.

State	Meaning
WAITING_STABLE	Waiting for initial or transition steady state.
RUNNING_STEP	Step applied and waiting for response.
WAITING_FINAL_STABLE	Response detected; waiting for final steady state.
FINISHED	Identification finished.
ABORTED	Test aborted.

StepTestError

Error	Reason
NONE	No error.
INVALID_CONFIG	Invalid configuration.
PV_LOW / PV_HIGH	PV outside test limits.
OP_LIMIT	Required OP outside test limits.
TIMEOUT	Maximum time exceeded.
NO_RESPONSE	No valid response detected.
INVALID_MODEL	A response occurred, but no valid model could be obtained.
USER_ABORT	Aborted by application/user.

RelayTestState

State	Meaning
IDLE	No active test.
RUNNING	Relay test running.
FINISHED	Ku/Tu calculated.
ABORTED	Test aborted.

RelayTestError

Error	Reason
NONE	No error.
INVALID_CONFIG	Incompatible parameters or reference out of range.
PV_LOW / PV_HIGH	PV outside limits.
OP_LIMIT	Bias or OP high/low outside limits.
TIMEOUT	Cycles exist but the required number is not completed in time.

Error	Reason
NO_OSCILLATION	Time expires without completing cycles.
USER_ABORT	Aborted by application/user.

Both StepTest and RelayTest return the controller to MAN and restore the original bias when they finish or are aborted. Reset() clears state, errors and results while preserving the object configuration. The global RESET command in the TCLab sketch, by contrast, explicitly reapplies its default values.

15. TCLab sketch: acquisition, filtering, safety and validation

The library has been validated on a TCLab with an Arduino UNO R4 WiFi. This sketch is not part of the algorithmic core; it is a test application that integrates hardware, safety, console commands and plotting.

Item	Configuration
T1	A0
T2	A2
Q1	D3
Q2	D5
Serial	115200 baud
ADC	14 bits, 3.300 V external reference
Q1 100 %	Maximum PWM 200/255
Q2	Maximum PWM 100/255, held at 0 in the current test
I/O period	100 ms
Plot period	500 ms

Acquisition chain

```

ADC
-> discard first conversion after channel change
-> average of 10 conversions
-> MedianFilter<5>
-> LowPassFilter(tau)
-> PV used by PID / StepTest / RelayTest

```

The sensor conversion follows the TMP36 relationship:

Temperature

$$T [C] = (V - 0.5) * 100$$

Operating range and thermal protection

Concept	Current value
Normal operating SP	10..80 °C
Trip on filtered PV/T2	>= 80 °C
Trip on raw PV/T2	>= 82 °C
Alarm reset	< 78 °C on T1 and T2
Action after trip	MAN, OP=0; does not automatically return to AUTO

TCLab thermal protection is independent of the StepTest and RelayTest limits. A final OP clamp is also applied before writing to the hardware.

16. TCLab sketch serial-command interface

Do not confuse API and console. The following commands are not part of the PIDControl core. They are a textual interface implemented by the TCLab sketch to test the library through Serial/Arduino COM Plotter.

General commands

Command	Function	Example
HELP	Shows help.	HELP
STATUS	Shows PID, filter and test status.	STATUS
RESET	MAN, OP=0 and restores sketch defaults.	RESET
STOP	Aborts tests, sets MAN and OP=0.	STOP
MAN [op]	Switches to MAN and optionally sets OP.	MAN 40
AUTO [sp]	Switches to AUTO and optionally sets SP.	AUTO 50
SP <valor>	Updates operator SP, limited to 10..80 °C.	SP 50
OP <valor>	Updates manual OP.	OP 35
TUNE <Kc> <Ki> <Kd>	Updates tuning.	TUNE 6.404957 0.042629 0
STRUCT PID PI_D I_PD	Selects structure.	STRUCT PI_D
ACTION DIRECT REVERSE	Selects action.	ACTION REVERSE
TRACK ON OFF	Enables/disables PV Tracking.	TRACK ON
OUTLIMITS <min> <max>	PID output limits.	OUTLIMITS 0 100
TS <ms>	Sample time.	TS 100
FILTER <tau>	Median(5) + LPF with specified tau.	FILTER 1.0
FILTER ON / OFF	Enables default LPF or disables it while keeping the median filter.	FILTER OFF
IDENT [STEP RELAY]	Reprints identification results stored in memory.	IDENT

StepTest commands

Command	Function
STEP START / STEP ABORT	Starts or aborts the configured test.
STEP SIZE <percent>	OP step size.
STEP N <1..10>	Number of steps.
STEP DIR UP DOWN	Direction of the first step.

Command	Function
STEP GAIN POS NEG	Expected sign of process gain.
STEP STABLE <band> <time> [slope]	Steady-state criterion.
STEP RESPONSE <deltaPV>	Minimum directional response threshold.
STEP TIME <minutes>	Maximum time.
STEP LIMITS <opMin> <opMax> <pvMin> <pvMax>	Test safety limits.
STEP RANGE <pvRangeMin> <pvRangeMax> <opRangeMin> <opRangeMax>	Full engineering ranges for normalized Kp.

Standard compact format:

```
STEP(size,n,UP|DOWN,POS|NEG,stableBand,stableTime,maxSlope,response,maxMin,opMin,opMax,pvMin,pvMax)
```

With optional engineering ranges:

```
STEP(...,pvMin,pvMax,pvRangeMin,pvRangeMax,opRangeMin,opRangeMax)
```

Validated example:

```
STEP(10,2,UP,POS,0.3,60,0.10,0.4,30,0,60,15,45,0,80,0,100)
```

RelayTest commands

Command	Function
RELAY START / RELAY ABORT	Starts or aborts the test.
RELAY AMP <percent>	Amplitude d around the bias.
RELAY HYST <degC>	Switching hysteresis.
RELAY CYCLES <n>	Number of valid cycles to average.
RELAY REF CURRENT	Uses current PV as the reference at start.
RELAY REF <degC>	Uses a fixed reference.
RELAY TIME <minutes>	Maximum time.
RELAY LIMITS <opMin> <opMax> <pvMin> <pvMax>	Safety limits.
RELAY(amp,hyst,cycles,CURRENT ref,maxMin,opMin,opMax,pvMin,pvMax)	Configures and starts RelayTest in one command.

Compact format in v1.4.2. The TCLab sketch interprets RELAY(...) and converts its parameters into RelayTestConfig. Test logic, validation and calculation remain inside PIDControl. STEP(...) keeps its existing compact format.

Compact RelayTest format

The compact format validated on TCLab is:

```
RELAY(10,0.5,3,CURRENT,15,20,60,15,48)
```

It is equivalent to a relay amplitude of ± 10 %OP, hysteresis of ± 0.5 °C, three valid cycles, reference equal to the current PV, a 15 min timeout, allowed OP of 20..60 % and allowed PV of 15..48 °C.

Retrieving results with IDENT

After a StepTest or RelayTest finishes, the calculated result remains stored in memory. If the serial-port log is cleared or lost, the result can be requested again without repeating the test:

```
IDENT
IDENT STEP
IDENT RELAY
```

IDENT shows K_p , T_0 and T_p from the stored StepTest, or K_u and T_u from the stored RelayTest. It also prints the available tuning proposals again. The memory is volatile: RESET, a board reset/power loss, or a new test of the same type removes or overwrites these data.

Frame for Arduino COM Plotter

```
#SP PV OP T2
```

The frame starts with '#', uses a space as the separator and contains four variables.

17. Recommended workflows

Normal operation

```
pid.SetOutputLimits(0.0, 100.0);
pid.SetSampleTime(100);
pid.SetStructure(PIDType::PI_D);
pid.SetDirection(PIDAction::REVERSE);
pid.SetPVTracking(true);

pid.SetManualOutput(40.0);
pid.SetMode(PIDMode::MAN);

// When appropriate:
pid.SetMode(PIDMode::AUTO);
pid.SetSetpoint(50.0);
```

FOPDT identification + IMC/Lambda

1. Stabilize the process in MAN.
2. Configure StepTest limits and ranges.
3. Run test.Start() and call test.Update() in loop().
4. Obtain FOPDTModel model = test.GetModel().
5. Calculate an IMC/Lambda PI proposal and a Lambda PID proposal.
6. Review Kp, T0, Tp and the consistency of the test.
7. Explicitly apply the selected tuning.
8. Switch from MAN to AUTO with PV Tracking.
9. If the TCLab sketch log is lost, use IDENT STEP while the result remains in memory.

Relay experiment

1. Stabilize the process in MAN with a suitable bias.
2. Choose amplitude and hysteresis.
3. Verify that bias+d and bias-d remain within limits.
4. Run relay.Start(config) or the compact RELAY(...) command.
5. Call relay.Update() and pid.Compute() in loop().
6. Obtain Ku and Tu.
7. Calculate Tyreus-Luyben PI and PID with PIDTuning.
8. The test restores the bias and remains in MAN.
9. If the TCLab sketch log is lost, use IDENT RELAY while the result remains in memory.

Safety. Identification tests should have their own limits that are more conservative than the absolute physical limits of the plant or setup. In addition, independent hardware/application protection must remain active.

18. Complete API reference

PIDControl

Method	Description
PIDControl(...)	Constructor with PV/OP/SP pointers, Kc/Ki/Kd, structure, action and tracking.
Compute()	Runs MAN/tracking or AUTO calculation as appropriate.
SetMode()/GetMode()	Changes/queries MAN or AUTO.
SetManualOutput()/GetManualOutput()	Sets/queries manual OP.
SetTunings()	Sets Kc, Ki and Kd; ignores negative parameters.
GetKc()/GetKi()/GetKd()	Returns parameters.
SetStructure()/GetStructure()	PID, PI_D or I_PD.
SetDirection()/GetDirection()	DIRECT or REVERSE.
SetPVTracking()/GetPVTracking()	Enables/queries tracking.
SetSetpoint()/GetOperatorSetpoint()	Updates/queries operator setpoint.
SetOutputLimits()	Configures OP limits.
GetOutputMinimum()/Maximum()	Queries limits.
SetSampleTime()/GetSampleTime()	Configures/queries calculation period in ms.

StepTest

Method	Description
SetOutputLimits()	OP limits during the test.
SetPVLimits()	PV limits during the test.
SetOutputRange()/SetPVRange()	Full ranges for normalization.
HasEngineeringRanges()	Indicates whether both ranges are configured.
SetStepSize()	Step size.
SetNumberOfSteps()	Number of steps, default maximum 10.
SetFirstStepUp()	Initial direction.
SetProcessGain()	POSITIVE/NEGATIVE.
SetStabilityCriteria()	Band, time and optional maximum slope.
SetMaxStabilitySlope()	Maximum slope.
SetResponseThreshold()	Response threshold.
SetMaxTestTime()/GetMaxTestTime()	Timeout in minutes.
Start()/Update()/Abort()/Reset()	Lifecycle.

Method	Description
GetState()/GetError()	State and error.
GetResult()	Aggregated result.
GetModel()	Transferable FOPDTModel.
GetStepResult()	Individual step result.
GetCurrentStabilitySlope()	Current stability-window slope.
GetBiasOutput()/GetCurrentTargetOutput()	Bias and target OP.
GetCurrentStepIndex()	Current step index.

RelayTest

Method	Description
SetOutputLimits()/SetPVLimits()	Test safety limits.
SetRelayAmplitude()	Amplitude d.
SetHysteresis()	PV hysteresis.
SetCycles()	Valid cycles to average.
SetReference()	Fixed reference.
UseCurrentPVAsReference()	Reference = PV at start.
SetMaxTestTime()/GetMaxTestTime()	Timeout in minutes.
Start()	Starts with previously stored configuration.
Start(const RelayTestConfig&)	Configures and starts in one call.
Update()/Abort()/Reset()	Lifecycle.
IsRunning()/IsFinished()/IsAborted()	State queries.
GetState()/GetError()/GetResult()	State, error and Ku/Tu.

PIDTuning

Method	Description
PIDTuning(model)	Constructor from FOPDTModel.
IMC_PI()	IMC-PI with default Tf.
IMC_PI(Tf)	IMC-PI with explicit Tf.
IMC_PI(IMCSpeed)	IMC-PI with Tf preset.

Method	Description
DefaultLambda()	Returns the default Tf; the name is kept for API compatibility.
GetModel()	Returns the stored model.
Apply(pid,result)	Applies Kc/Ki/Kd from a valid result.
Lambda_PI() / Lambda_PI(Tf) / Lambda_PI(IMCSpeed)	Explicit alias for the IMC/Lambda PI rule.
Lambda_PID() / Lambda_PID(Tf) / Lambda_PID(IMCSpeed)	Calculates Lambda PID from FOPDT and returns Kc, Ti, Td, Ki and Kd.
TyreusLuybenPI(Ku,Tu)	Calculates Tyreus-Luyben PI from Ku and Tu.
TyreusLuybenPID(Ku,Tu)	Calculates Tyreus-Luyben PID from Ku and Tu.

PIDControlFilters

Class	Main methods
MovingAverageFilter<N>	Reset(), Reset(x0), Update(x), Value(), IsInitialized(), Count(), WindowSize().
MedianFilter<N>	Reset(), Reset(x0), Update(x), Value(), IsInitialized(), Count(), WindowSize().
LowPassFilter	SetTimeConstant(), GetTimeConstant(), Reset(), Update(x,dt), Value(), IsInitialized().
ComplementaryFilter	SetAlpha(), GetAlpha(), Reset(), Update(), UpdateIntegrated(), Value(), IsInitialized().

19. Limitations and good practices

- The controller does not know the plant physics or units; PV and OP scaling belongs to the application.
- Process protection must be implemented outside the PID and must take priority over the calculated output.
- StepTest assumes an approximately monotonic response that can be represented by a FOPDT model in the tested operating region.
- Integrating, strongly oscillatory, inverse-response or highly nonlinear processes may require a different identification method.
- The ProcessGain sign in StepTest must match the physical OP→PV relationship.
- RelayTest infers the process sign from PIDAction: REVERSE is treated as positive and DIRECT as negative.
- Relay hysteresis that is too small may switch because of noise; hysteresis that is too large distorts the experiment.
- Excessive filtering changes the observed dynamics and may increase T0.
- Changes to tuning, structure, action or limits in AUTO are bumpless, but the process should still be supervised after an important change.
- Identification results are valid for the operating region where the experiment was performed; they do not guarantee global linearity.

About model quality. A valid model at the end of StepTest means that the algorithm was able to calculate one. In demanding applications, compare several steps or tests and evaluate the repeatability of Kp, T0 and Tp before treating a tuning as final.

20. Package structure and repository preparation

```
PIDControl/
|-- library.properties
|-- keywords.txt
|-- README.md
|-- CHANGELOG.md
|-- src/
|   |-- PIDControl.h / .cpp
|   |-- StepTest.h / .cpp
|   |-- RelayTest.h / .cpp
|   |-- PIDTuning.h / .cpp
|   `-- PIDControlFilters.h
|-- examples/
|   |-- BasicPID/
|   |-- ManualAuto/
|   |-- StepTestIdentification/
|   |-- StepTestAnalogIO/
|   |-- RelayAutoTune/
|   |-- IMCTuning/
|   `-- FilteringPV/
|-- docs/
`-- extras/
```

Installation

PIDControl is available in Arduino Library Manager and can be installed directly from the Arduino IDE library manager by searching for `PIDControl`. It can also be installed manually by downloading the ZIP from GitHub or the project page and selecting Sketch > Include Library > Add .ZIP Library.

- <https://github.com/deurus/PIDControl>
- <https://garikoitz.info/blog/descargas/PIDControl/PIDControl-1.4.2.zip>

Appendix A. Complete minimal example

```
#include <PIDControl.h>

double SP = 50.0;
double PV = 0.0;
double OP = 0.0;

const bool pvTracking = true;

PIDControl pid(
    &PV, &OP, &SP,
    1.0, 0.4, 0.0,
    PIDType::PI_D,
    PIDAction::REVERSE,
    pvTracking
);

void setup()
{
    pid.SetSampleTime(100);
    pid.SetOutputLimits(0.0, 100.0);
    pid.SetManualOutput(35.0);
    pid.SetMode(PIDMode::MAN);
}

void loop()
{
    PV = ReadPV();

    pid.Compute();

    WriteOP(OP);
}
```

The example deliberately leaves out `ReadPV()` and `WriteOP()`. This separation is part of the design: PIDControl works in engineering units and is not tied to a specific sensor or actuator type.

Appendix B. Quick reference

Need	Class / function
Control a PV	PIDControl
Work in manual/automatic	SetMode(), SetManualOutput()
Avoid MAN→AUTO steps	PV Tracking + bumpless
Change structure	PIDType::PID / PI_D / I_PD
Change control direction	PIDAction::DIRECT / REVERSE
Avoid windup	Internal conditional integration
Filter noise	PIDControlFilters
Identify Kp/T0/Tp	StepTest
Normalize gain	SetPVRange() + SetOutputRange()
Tune PI using IMC	PIDTuning
Obtain Ku/Tu	RelayTest
Configure RelayTest in one call	RelayTestConfig + Start(config)
Obtain PI from FOPDT	PIDTuning::IMC_PI() / Lambda_PI()
Obtain PID from FOPDT	PIDTuning::Lambda_PID()
Obtain PI/PID from Ku/Tu	PIDTuning::TyreusLuybenPI()/PID()
Retrieve sketch identification	IDENT / IDENT STEP / IDENT RELAY

With this architecture, a project can go from a noisy sensor reading to a tuned controller without coupling the control logic to specific hardware.