PID-Controlled Soldering Iron Temperature Using NTC Thermistor

Overview

This document captures the full technical aspect of building a **PID-Controlled Soldering Iron Using a NTC Thermistor**. It not only explains what was built and how, but also reflects on all the struggles, discoveries, and learning that happened along the way. This record is designed to serve as a future reference and a foundational piece in my embedded systems knowledge.

1. Introduction to the Project

The aim was to build a **closed-loop temperature control** system for a soldering iron. The desired setpoint was **100°C,** and the system used an **optocoupler** based **relay** to control the heat of the soldering iron. An **NTC thermistor** was used as a temperature sensor. The core of the control system was a **PID controller**.

2. What is PID Control? (Theory)

PID = Proportional + Integral + Derivative

It's a feedback control algorithm used to bring a system's output (like temperature) as close as possible to a target (setpoint) by minimizing the error between them.

Imagine you're trying to keep your soldering iron at exactly **100°C**. If the current temperature is **below 100°C**, the controller needs to **increase heat**. If it's **above**, it should **reduce or stop the heating**.

🔧 **Here's what each part of PID does:**

✅ **Proportional (P) – Reacts to *Present* Error**

- **Definition**: Applies a correction **proportional to how far off you are** from the setpoint.
- **Behaviour**: Bigger the error, stronger the correction.

Example:
If the iron is at **90°C** and you want **100°C**, the error is **+10**.
A proportional controller might say:
"Push more power because we're far below target!"

But... if Kp (proportional gain) is too high, it may **overshoot** to 120°C.

✅ **Integral (I) – Reacts to *Past* Error**

- **Definition**: Adds up all past error over time to eliminate **residual offset**.
- **Behaviour**: Helps if you're **always slightly off**, even after P tries to fix it.

Example:
The iron stays stuck around **98°C**, never quite reaching 100°C.
Proportional may stop reacting because error is small.
Integral slowly **accumulates that 2°C error** and pushes the system further, until it reaches the setpoint.

⚠️ Too much I → causes "wind-up" → the system can overshoot and oscillate.

✅ **Derivative (D) – Reacts to *Future Trend***

- **Definition**: Predicts **how fast** the error is changing.
- **Behaviour**: Tries to **prevent overshoot** by braking early.

Example:
If you're at **96°C**, but the temp is **rising quickly**, D says:
"Slow down, we're approaching the target too fast!"
It dampens the response, especially during sharp increases.

⚠️ Too much D can cause sluggishness or react to noise.

The goal of PID is to compute an output (in this case, heater ON time) that reduces the difference between the current temperature and the target.

3. When PID Theory Meets Reality

On paper, PID control looks mathematically perfect.
In simulation, it responds smoothly, converges to the setpoint, and eliminates error like a dream.

But the real world?
It's messy, slow, noisy, and full of quirks that textbooks don't warn you about.

Practical Challenges I Faced:

---

1. The Relay Isn't Analog, It's Just ON or OFF

In theory, **PID** outputs a smooth range of values (e.g., 0 to 255) to control things proportionally.

But your relay isn't analog.
It doesn't understand "a little ON" — it's either fully ON or fully OFF.

So, to apply **PID** output, you had to translate analog PID output into timed ON/OFF pulses (a technique called **time-proportional control**).
For example:

- If **PID** gives **60%,** turn the relay **ON for 600ms**, **OFF for 400ms** in a 1-second window.

Problem: If the **PID** output changes frequently, the relay keeps switching rapidly — which it's not designed for.

That's why you added logic to limit relay switching frequency — a smart move for safety and longevity.

---

2. Thermistors Are Noisy, Slow, and Placement-Sensitive

You used an **NTC thermistor** module to read temperature. But thermistors:

- Are slow to react to rapid temperature changes.

- Are sensitive to placement (a few mm away can cause 10°C error).
- Are noisy — slight **ADC** jitter can look like fluctuations.

You placed it on or near the soldering iron tip, but:

- If not perfectly attached, it lags behind real tip temperature.
- Any air gaps or loose contact led to false readings.
- Even touching the soldering iron can heat the thermistor locally, not the tip itself.

All of this made tuning difficult because the "temperature" the PID was reacting to wasn't always accurate or consistent.

---

3. Thermal Inertia – Heat Doesn't Instantly Change

This is a big real-world difference from simulations.

Your soldering iron has thermal mass — it:

- **Takes time to heat up after applying power.**
- **Continues heating even after power is cut, due to stored heat.**
- **Cools slowly, unless externally cooled.**

This leads to overshooting.
Imagine:

1. **Thermistor** reads 98°C.
2. **PID** says "apply full power!" → relay turns ON.
3. Iron starts heating — but **thermistor lags.**
4. By the time thermistor reads **100°C**, the iron is already at **110°C** internally.

This thermal inertia makes PID tuning tough.
Even a perfectly tuned system on paper can oscillate wildly in practice.

---

4. Tuning Took Forever – Even with AutoTune

Initially, you did manual hit-and-trial tuning:

- You tried random values of Kp, Ki, Kd.
- You observed how fast it heats, how much it overshoots.
- And then adjusted again... and again... and again.

This was frustrating and time-consuming.

Later, when you discovered **PID_AutoTune,** things got better — but not instantly:

- It oscillated the output intentionally, to learn how the system responds.
- This took several minutes, during which you had to wait patiently, watching.
- It felt odd — the system was deliberately being pushed out of control — but it was necessary for learning.

Even after **AutoTune** gave values, you still had to fine-tune them for stability.

Bottom Line: Real Control is a Dance

The goal wasn't just to hit 100°C.
The goal was to reach it ***smoothly*****,** ***quickly*****,** **and** ***without overshoot*** — again and again.

You were not just writing code — you were:

- Dealing with thermal physics.
- Interfacing with noisy sensors.
- Managing relay limitations.
- Applying feedback theory in a noisy, laggy environment.

And despite all this, you got it to work.
That's what makes embedded control satisfying — when your system *finally* behaves like you intended.

---

4. Hardware Setup

Microcontroller used: ESP32

- **Sensor:** 10k NTC thermistor
- **Analog pin:** GPIO34 (ADC input) for **thermistor module**
- **Relay module:** ON/OFF heater control **(GPIO4)**
- **AutoTune button**: **GPIO2**
- **Controller**: ESP32 board

The thermistor was used in a voltage divider. Its resistance drops as temperature rises. The voltage was read via ADC and mapped to approximate temperature.

5. Initial Attempts and Struggles

You first implemented a PID controller using the **PID_v1.h library**. The logic was simple:

- Read temperature
- Compare it to setpoint **(100°C)**
- Calculate **PID** output
- If **output** > **threshold** → Turn relay ON
- Else → **Turn relay OFF**

But here came the first big problem: **overshoot**.

Symptoms:

- Temperature went up to **120°C** or more.
- Then it dropped slowly, sometimes down to **50°C**.
- Then heated up again — causing a saw-tooth wave.

Root Causes:

- Poor **PID tuning**
- Heater's thermal lag
- Relay can't do precise control like a **MOSFET**

I spent a lot of time using trial-and-error to adjust **Kp, Ki, and Kd.** But each small change led to wildly different behavior. There was no stability.

6. Discovery of PID AutoTune

Then came a major breakthrough: **PID_AutoTune_v0.h**.

You wired a button to start the auto-tuning process.
When you pressed it:

- The system turned the heater ON and OFF repeatedly.
- It watched how fast the temperature rose and fell.
- It created oscillations in the system on purpose.

What are Oscillations?
It means the temperature is made to fluctuate up and down around the setpoint.
AutoTune uses this behavior to calculate optimal PID constants.

This gave you more reasonable values for **Kp, Ki, and Kd.**

7. Time Proportional Control (Relay + PID)

Another insight you had: **PID** output was a number, not just **ON/OFF**.
But relay can only switch ON or OFF, not produce analog output.

So, you used time-proportional control:

```
if (millis() - windowStartTime > WindowSize)
    windowStartTime += WindowSize;

if (Output > millis() - windowStartTime)
    digitalWrite(RELAY_PIN, HIGH);
else
    digitalWrite(RELAY_PIN, LOW);
```

This code divides time into windows (e.g., 2000 ms). If PID says "**60% power**", the relay stays ON for 1200 ms, then OFF for 800 ms.

This mimics analog control with a relay.

8. Remaining Challenges

- NTC mapping was crude (not using Steinhart-Hart yet)
- Relay still clicks frequently, not ideal for long-term use
- Thermistor still noisy → needs digital filtering
- PID still needs minor returning based on environment

9. Achievements

- **Stable Temperature Control at ~100°C for a Soldering Iron**
  Achieving ~100°C on a soldering iron isn't like heating water. You dealt with **thermal inertia**, **overshoot**, and **sensor lag** — yet still maintained a steady temperature with **minimal oscillation**.
- **Implemented and Understood Practical PID Control**
  You didn't just "use a PID library." You understood:
  - What each of **Kp, Ki, Kd** actually *does* in real systems.
  - How they behave **differently** than in theory (due to delays, inertia, and sensor placement).
  - Why PID isn't a magic wand — it's a system that needs to be shaped for the real world.
- **Discovered and Used PID AutoTune Efficiently**
  You initially tuned manually (trial-and-error), and later discovered **PID_AutoTune** — which you learned to use properly despite its odd behaviour and long wait times. This saved time, gave more consistent gains, and improved overall performance.
- **End-to-End Embedded Control Pipeline Completed**
  From **sensor reading → PID processing → relay actuation → real-world thermal response**, you created and completed a **closed-loop embedded system**. That's what real embedded engineers do.
- **Hands-On Learning That Changed Your Understanding of Control Systems**
  This project transformed how you view "control" — not just as a block diagram, but as a **living system** affected by inertia, lag, noise, and limitations. That insight will stay with you in every future robotics, drone, or thermal project.

10. Lessons for the Future

- PID needs patience and understanding
- **AutoTune** saves time but should be understood
- Optocoupler Relays are okay to start but not ideal for precision (may wear out in long run if used so frequently, to avoid losses due to wear and tear in frequently changing environments a **SSR** (Solid-Stae Relay) is used. But if using only **optocoupler based relays**, some sort of delay is needed (left for future debugging))
- Sensor accuracy and mapping matter a lot
- Logging your values is critical to debugging

Final Thought

This project wasn't just about building a controller. It taught me about:

- This wasn't just about building a temperature controller — it was a **masterclass in real-world embedded control**. I learned:
- 🧠 **Control Systems, Beyond the Textbook**
  I didn't just apply a PID formula — I understood what control means in the presence of **noise**, **delay**, **inertia**, and **binary actuators**. This is where theory meets reality.
- 📉 **Sensing the Real World is Messy**
  NTC thermistors aren't perfect. I dealt with:
- **Slow thermal response**
- **Placement sensitivity**
- **Noisy ADC readings**
  And still, I extracted usable, stable feedback from them. That's engineering.
- 🔁 **Everything is a Loop, Nothing is Instant**
  From heating lag to cooling delay, I grasped **system inertia**. It taught me to think in time, not just in steps — a vital skill for robotics, power electronics, and flight systems alike.
- 🎛️ **Tuning is Art + Math**
  Manual tuning taught me *patience and intuition*. AutoTune taught me *structure*. Both taught me that no simulation prepares you for real dynamics.
- ⚡ **Digital Systems Aren't Analog**
  Using a **relay** instead of a DAC meant thinking in **time-based control**, not voltage. This built intuition for PWM, SMPS, and all digital modulation techniques I'll use in the future.
- 🧩 **Embedded Systems Are End-to-End Pipelines**
  Sensor → Algorithm → Actuator → World → Sensor again. I experienced the **full feedback loop**, and learned where bottlenecks, delays, and non-idealities creep in.
- 🌍 **Data Sheets Don't Warn You About the Real World**
  I learned that a spec value (like ±1% thermistor tolerance) means little when the soldering iron radiates heat sideways, or when ambient temperature fluctuates. The real world has **leaks, noise, and inertia** — and control must embrace that.
- 🎯 **Engineering is Controlled Imperfection**
  The goal isn't perfection — it's stability, reliability, and adaptability. I saw how a "stable" system doesn't mean "dead flat," but "oscillates within tolerance, and responds predictably."

Code Snippets

```
#include <PID_v1.h>
#include <PID_AutoTune_v0.h>

#define THERMISTOR_PIN 34       // ADC pin for temperature input
#define RELAY_PIN 4             // Output pin to control relay
#define BUTTON_PIN 2           // Button to trigger AutoTune

unsigned long autoTuneLastSwitch = 0;
const unsigned long autoTuneRelayCycle = 3000;  // 3 seconds relay
ON/OFF interval

unsigned long lastPrint = 0;

// === PID variables ===
double Setpoint = 100.0;
double Input = 0.0;
double Output = 0.0;
double Kp = 2.0, Ki = 5.0, Kd = 1.0;

// === PID and AutoTune objects ===
PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);
PID_ATune aTune(&Input, &Output);

const unsigned long windowSize = 2000;  // Relay time-proportional
window
unsigned long windowStartTime = 0;

bool autoTuneRunning = false;
bool lastButtonState = HIGH;

double readTemperature() {
  int adc = analogRead(THERMISTOR_PIN);
```

```
  if (adc <= 0 || adc >= 4095) return -273.15;  // prevent divide-by-zero or infinite

  // For pull-down configuration (thermistor on top)
  double resistance = 10000.0 * adc / (4095.0 - adc);  // Corrected formula

  // Steinhart-Hart calculation
  double steinhart = resistance / 10000.0;  // R/R0
  steinhart = log(steinhart);               // ln(R/R0)
  steinhart /= 3950.0;                      // 1/B * ln(R/R0)
  steinhart += 1.0 / (25.0 + 273.15);       // + 1/T0
  steinhart = 1.0 / steinhart;              // Invert
  steinhart -= 273.15;                      // Kelvin to Celsius

  return steinhart;
}

void setup() {
  Serial.begin(115200);
  pinMode(RELAY_PIN, OUTPUT);
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  windowStartTime = millis();

  myPID.SetOutputLimits(0, windowSize);
  myPID.SetMode(AUTOMATIC);

  Serial.println("System Ready. Press button to start AutoTune.");
}

void loop() {
  Input = readTemperature();  // Replace with real sensor logic

  // === Button press detection === N
  // === Button press detection with toggle ===
  bool currentButtonState = digitalRead(BUTTON_PIN);
  if (lastButtonState == HIGH && currentButtonState == LOW) {
    if (!autoTuneRunning) {

      if ((now - windowStartTime) < Output) {
```

```
          digitalWrite(RELAY_PIN, HIGH);
        } else {
          digitalWrite(RELAY_PIN, LOW);
        }
      } else {
        digitalWrite(RELAY_PIN, LOW);  // Keep relay OFF during
AutoTune
      }

      aTune.SetOutputStep(50);        // Current step size is okay
      aTune.SetControlType(1);        // PID mode
      aTune.SetLookbackSec(20);       // Already set
      aTune.SetNoiseBand(1);          // Already set
      aTune.SetOutputStep(50);        // Reduce step if overshoot too
fast




      // START AUTOTUNE
      autoTuneRunning = true;
      myPID.SetMode(MANUAL);

      aTune.SetNoiseBand(1.0);
      aTune.SetOutputStep(500);
      aTune.SetLookbackSec(10);
      aTune.SetControlType(1); // PI

      Serial.println("🔁 AutoTune STARTED");
    } else {
      // STOP AUTOTUNE
      autoTuneRunning = false;
      myPID.SetMode(AUTOMATIC);

      Serial.println("⛔ AutoTune STOPPED manually.");
    }
  }
  lastButtonState = currentButtonState;
```

```
unsigned long now = millis();
if ((now - windowStartTime) > windowSize) {
  windowStartTime += windowSize;
}

// === AutoTune Mode ===
if (autoTuneRunning) {

  if (aTune.Runtime()) {
    // AutoTune complete — apply tuned values
  }

  // Only control relay ON/OFF every few seconds:
  if ((now - autoTuneLastSwitch) > autoTuneRelayCycle) {
    autoTuneLastSwitch = now;

    if (Output > 0) {
      digitalWrite(RELAY_PIN, HIGH);
    } else {
      digitalWrite(RELAY_PIN, LOW);
    }
  }

  if (aTune.Runtime()) {
    // AutoTune finished!
    autoTuneRunning = false;

    // Fetch tuned values
    Kp = aTune.GetKp();
    Ki = aTune.GetKi();
    Kd = aTune.GetKd();

    myPID.SetTunings(Kp, Ki, Kd);
    myPID.SetMode(AUTOMATIC);  // resume normal PID

    // Inform the user
    Serial.println("AutoTune finished!");
    Serial.print("Kp: "); Serial.println(Kp);
    Serial.print("Ki: "); Serial.println(Ki);
```

```
      Serial.print("Kd: "); Serial.println(Kd);

      Serial.println("==== AutoTune COMPLETE ====");
      Serial.println("👉 Copy these values into your code:");
      Serial.print("double Kp = "); Serial.print(Kp);
Serial.println(";");
      Serial.print("double Ki = "); Serial.print(Ki);
Serial.println(";");
      Serial.print("double Kd = "); Serial.print(Kd);
Serial.println(";");
      Serial.println("================================");

      Serial.println("AutoTune complete ☑ — Now resuming normal PID
control.");
      Serial.println("// 🔄 You may edit the Kp, Ki, Kd values below
if tuning wasn't ideal.");
    } else {
      if (millis() - lastPrint > 1000) {
        Serial.println("AutoTune running... please wait");
        lastPrint = millis();
      }
    }
  } else {
    myPID.Compute();
  }

  // === Time-proportional relay control ===
  if ((now - windowStartTime) < Output) {
    digitalWrite(RELAY_PIN, HIGH);
  } else {
    digitalWrite(RELAY_PIN, LOW);
  }

  // === Debug output ===
  if (!autoTuneRunning) {
    Serial.print("Temp: ");
    Serial.print(Input);
    Serial.print("°C | Output: ");
    Serial.print(Output);
```

```
    Serial.print(" | Relay: ");
    Serial.println(digitalRead(RELAY_PIN) ? "ON" : "OFF");
  }
  delay(200);
}

// === TEMP SENSOR MOCK (replace with real thermistor calc) ===
```