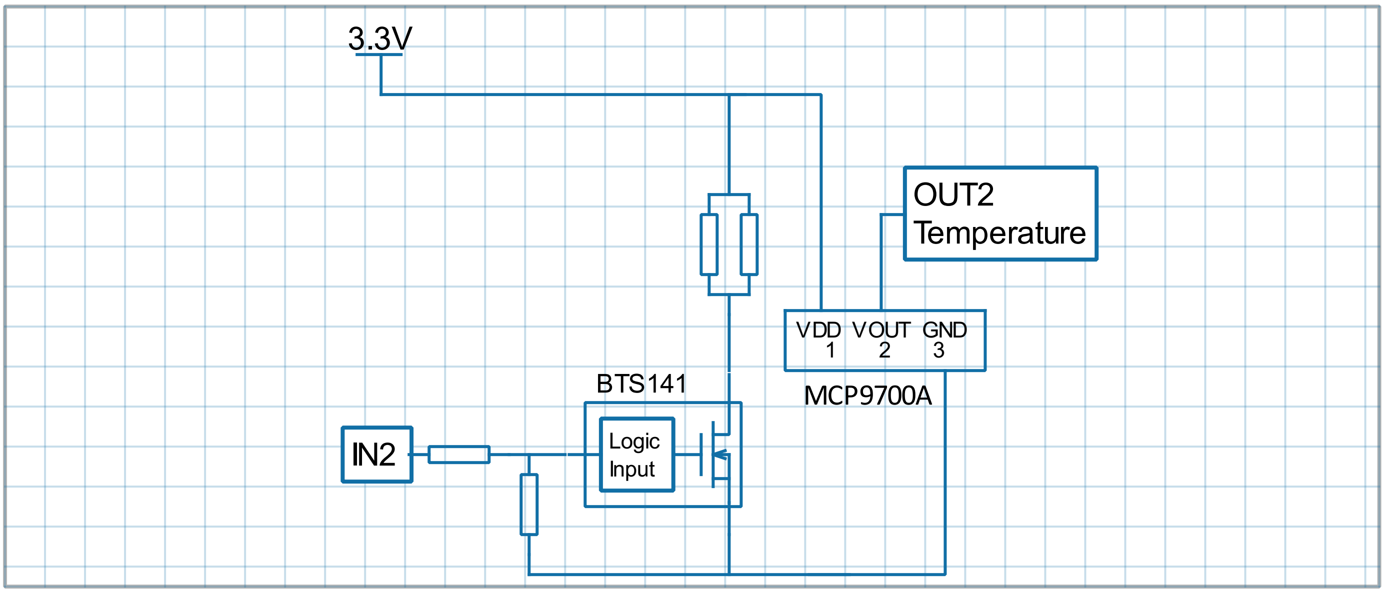
Temperature Control using LabJack and MatDeck's PID controller, with GUI Configuration

This document illustrates how the LabJack T7 device can be used to control ambient temperatures by switching the electronic circuit on and off. When the circuit is switched on, the current through the resistors causes it to heat up. Temperature measurement is performed by using MCP9701A temperature sensors. The LabJack T7 device is used to switch the circuit on and off using a digital output (DIO in PWM mode) and to measure the temperature using an analog input AIN. The PWM signal duty cycle, is set using a PID controller which is implemented in MatDeck. MatDeck provides GUI forms for effective and intuitive configuration of LabJack devices, as illustrated here.

Schematics of the electronic circuits

The schematics of the system described above for temperature control is displayed below. It should be pointed out that the schematics are created in MatDeck, which is suitable for various professional drawings.



The description of the circuit is as follows:

Functionality

- IN2 and OUT2 demonstrate PID temperature control

Demo board schematic pin descriptions

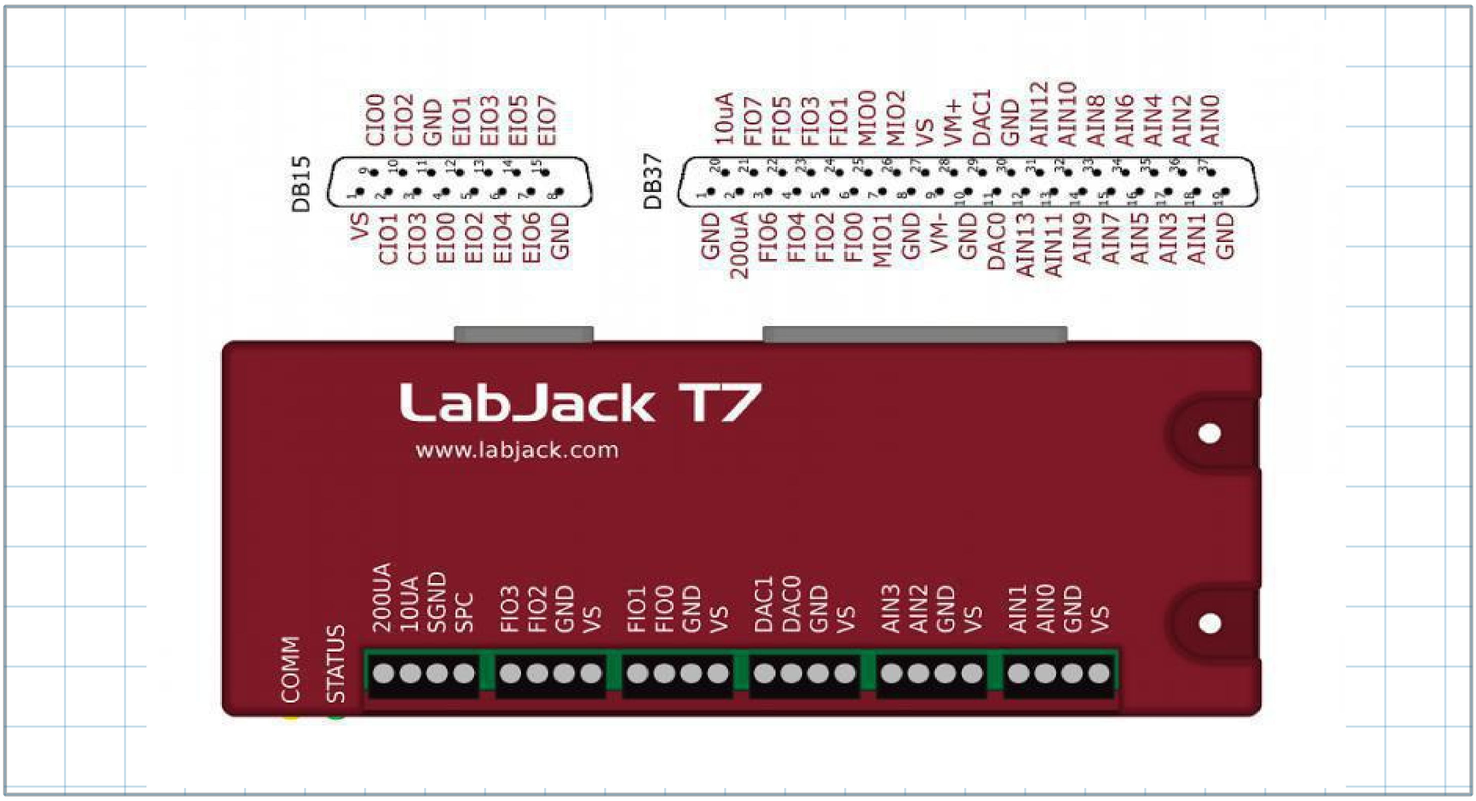
- IN2 input - connected to the PWM driver output in order to heat up the resistor
- OUT2 - output from the temperature sensor in mV

Connection to LabJack unit

- IN2 is connected to the PWM output
- OUT2 is connected to Analog inputs

Parts:

- For the power driver, BTS 141 is used which is a logic level low side driver.
- The temperature sensor used is MCP9700A-E/TO



Use of LabJack T7

In this experiment, the LabJack T7 device is used to produce a digital PWM output which is connected to the IN2. At the same time, T7 is used to measure the temperature by collecting the OUT2 signal at the AIN2 channel. MatDeck supports LabJack functions which can be used directly inside MatDeck's script to configure LabJack devices and to generate and acquire signals from the electronic circuits as described above. Here, details about the configuration of the selected features in this experiment are explained.

MatDeck also provides Graphical User Interface (GUI) plug-ins for simple, effective configuration. There are three plug-ins for three different groups of pins: ljdioT7_config_form() is used to configure DIOs, ljainT7_config_form() is used to configure AINs, and ljdac_config_form() is used to configure DACs. The details about the configuration of the selected features in this experiment are explained below.

GUI Configuration of DIO EF PWM out

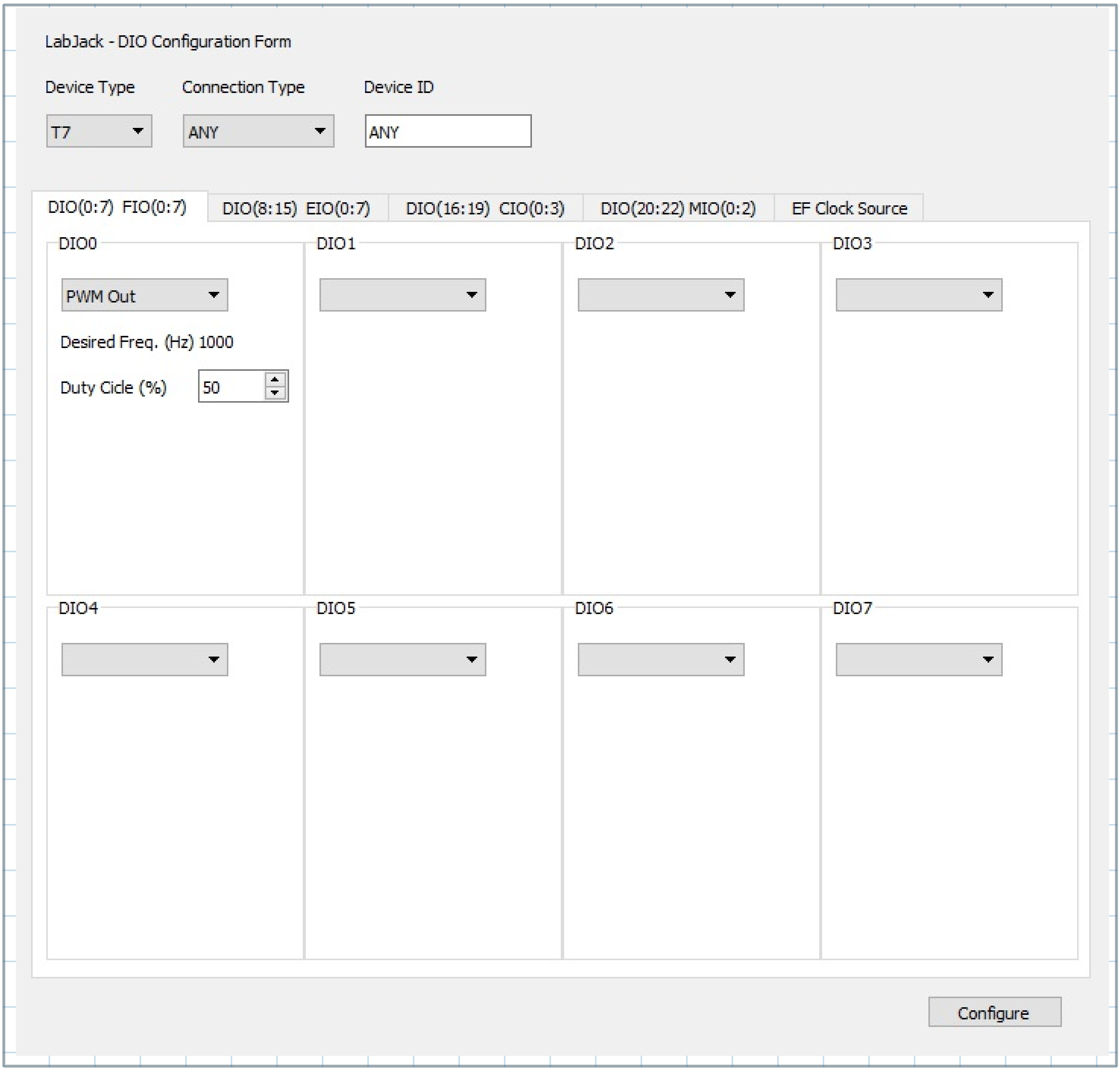
Here, DIO0 is used to produce the IN2 signals. When DIO0 is high, the transistor is switched on and the current through the resistors heats the temperature sensor. If DIO0 is low, the transistor is switched off and there is no current, thus the temperature will fall.

The GUI form for DIO configuration can be started using by ljdioT7_config_form(), as follows. The form is embedded within the canvas and used for the configuration of DIOs.

```
f := ljdioT7_config_form(0, "DIO form1")
ljdioT7_config_form_configure(f)
```

PWM Out at FIO0(DIO0) requires the clock source, thus the clock is first configured. There are three parameters to select for the configuration: clock source, clock divisor and the roll value for the given clock.

There are three different clocks supported by the T7, the most common is clock0 whose frequency is 80MHz. The clock divisor can be any power of two from 1, 2, up to 256, in this example we select a value of 1. The roll value is determined according to the desired frequency of the PWM Out signal. For example, if the desired frequency is 1kHz, the roll value is 80Hz/Divisor/1kHz=80000. In the GUI, it is possible to choose and set the desired frequency or desired roll value. PWM output at FIO0 (DIO0) is configured by selecting the appropriate option from the drop down menu. In the GUI, it is possible to set the desired value of the duty cycle directly to 50%.



LabJack - DIO Configuration Form

Device Type: T7

Connection Type: ANY

Device ID: ANY

DIO0: PWM Out

Desired Freq. (Hz) 1000

Duty Cicle (%) 50

GUI Configuring Temperature Measurement

The temperature is measured by using the AIN2 channel of the T7, where the OUT2 signal is connected. The low power linear active thermistor circuit ,MCP9701A, is used as a temperature sensor. Here, OUT2 is the voltage that depends on the ambient temperature, which should be converted into temperature using the linear function given in the data-sheet. The sensor transfer function is:

$$V_{OUT} = T_C \cdot T_A + V_{0°C}$$

Here, V_{OUT} is the sensor output voltage, T_A is ambient temperature, T_C is the temperature coefficient, and $V_{0°C}$ is the sensor output voltage at 0°C. From the MCP9701A datasheet, T_C=19.5 mV/°C and $V_{0°C}$=400mV. In order to determine the temperature from the voltage, we need the inverse of the function.

$$T_A = V_{OUT}/T_C - V_{0°C}/T_C$$

Slope and offset can be determined as follows:

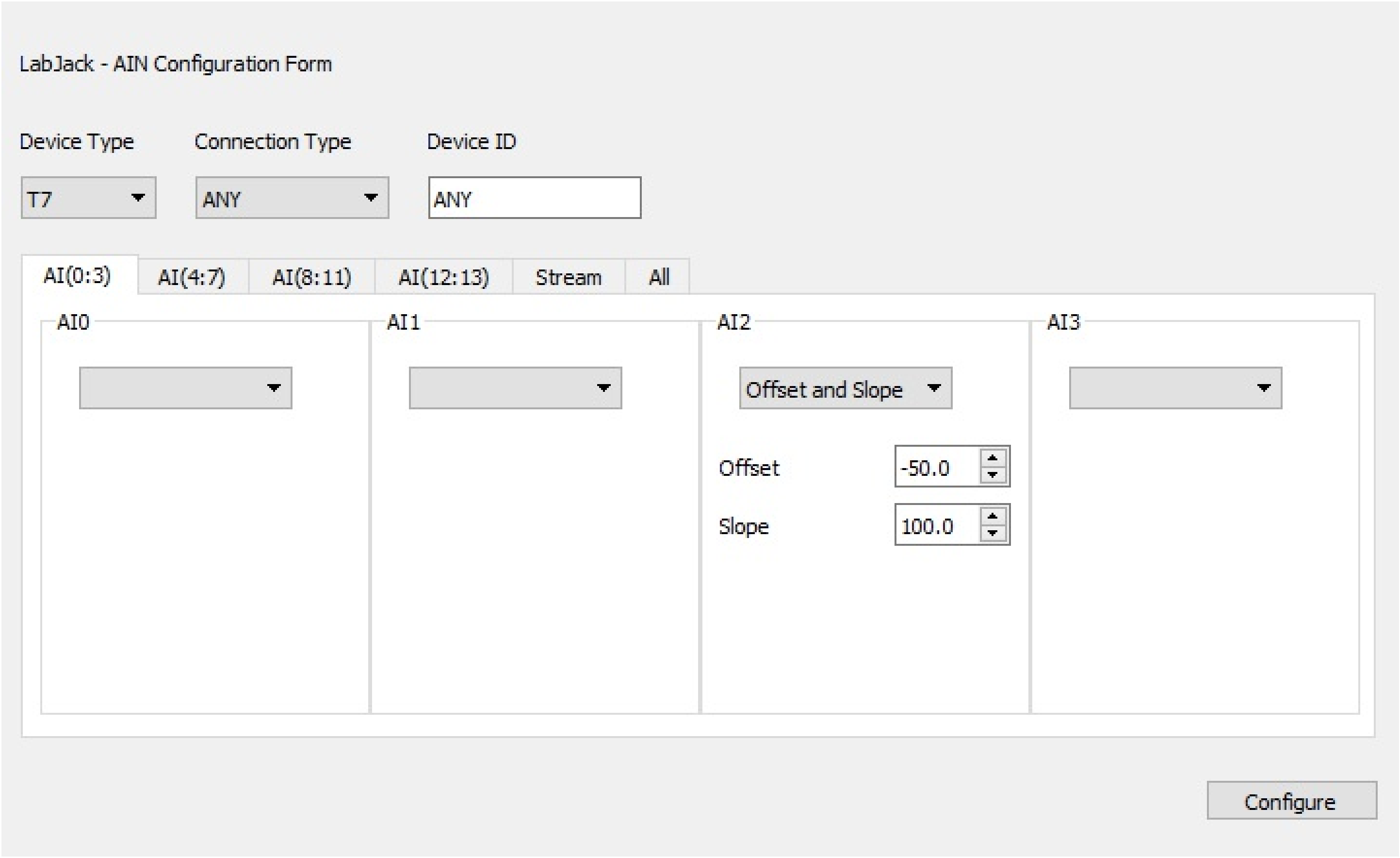
```
Tc := 0.0100
V0 := 0.5
Slope := 1 / Tc
Offset := -V0 / Tc
```

AIN2 is configured to use the Offset and Slope extended feature, EF_INDEX is 1, which automatically adds a slope and an offset to analog readings according to the linear function above.

$$\text{Slope} = 100 \qquad \text{Offset} = -50$$

MatDeck provides ljainT7_config_form() that can be used to set all the parameters graphically, which is very convenient for the user. In the following segment, there is an illustration on how to use the AIN configuration form. At the beginning, the form is evoked by calling the function ljainT7_config_form(). The form is embedded within the canvas and used for the configuration of AINs.

```
f2 := ljainT7_config_form(0, "AIN form1")
ljainT7_config_form_configure(f2)
```



LabJack - AIN Configuration Form

Device Type: T7

Connection Type: ANY

Device ID: ANY

AI2: Offset and Slope

Offset: -50.0

Slope: 100.0

Use of Configured LabJack T7

In order to use the configuration and use the device, the LabJack T7 device should be opened in the document:

```
dev := ljdevice_open("any", "any", "any")
```

The temperature is automatically read using:

```
Ta := ljdevice_read(dev, "AIN2_EF_READ_A")
```

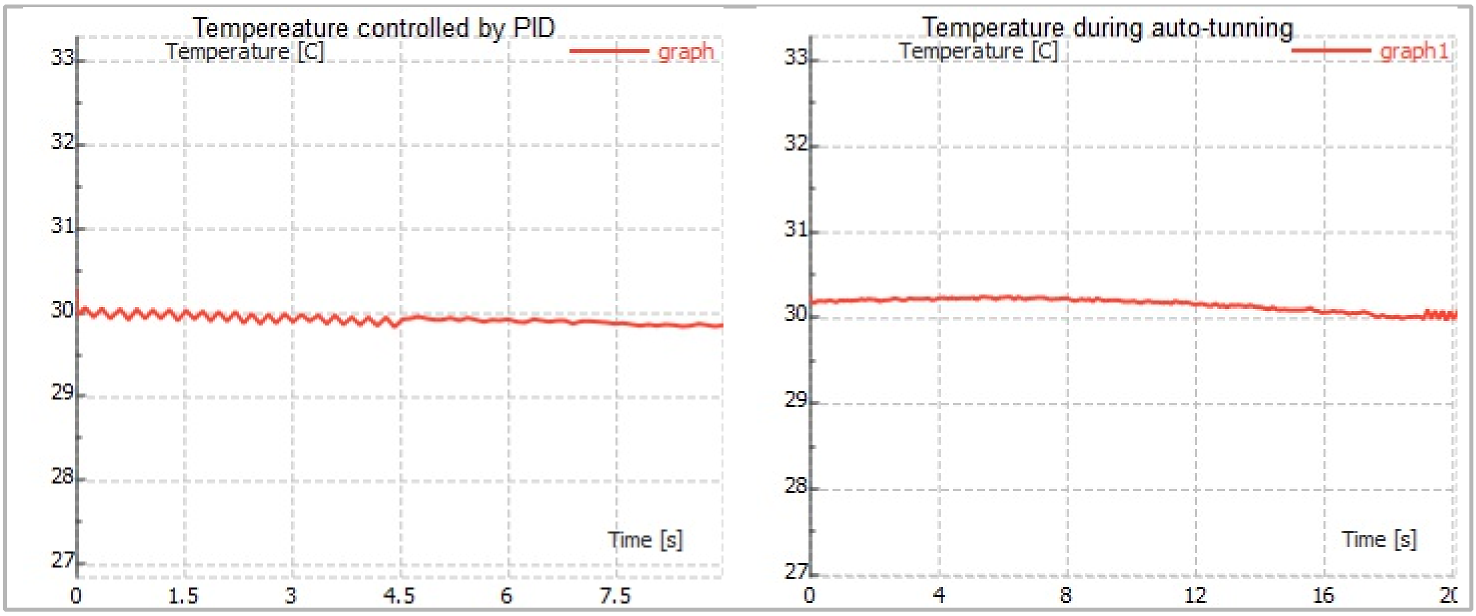
```
Ta = 29.861     C
```

PID temperature control

MatDeck provides a PID controller which can adjust the ambient temperature close to the sensor by switching the transistor on and off in the circuits shown in the schematics from above. MatDeck's PID controller is used in real-time operation to heat the system. The PID controller is a three component controller having proportional, integral and derivative terms. A PID controller continuously calculates the error value as the difference between a desired setpoint temperature and a measured temperature from the AIN2 LabJack channel. It then applies a correction based on the proportional, integral, and derivative terms using control variables. In the proposed example, the control variable is the duty cycle of the PWM out signal at DIO0. The user specifies the clock value which is the PID exchange period and the set point which is related to the controlled system. The PID controller widget contains the graph where it is possible to see the value of the measured process variable against the set point.

Auto-tuning PID controllers adjust its control parameters (proportional gain, integral gain, derivative gain) to the optimum values for the desired control response. The Ziegler–Nichols method for auto-tuning is the most common method in practice. The auto-tuning is based on the Ziegler–Nichols method, with the following different controllers: p, pi, pid, less overshoot, no overshoot, and Pessen integral.

```
graph := vector_create(2, true, 0) //Graph to show the Tempereature
graph[1] = Ta
graph1 := graph // Graph to show auto-tunning process
pid_period := 100
Target_T := 30
tunning_m := 2
// Call autotunning function
KPID := pid_autotune_direct(pid_period, Target_T, tunning_m, dev, 1, 1)
// Obtained proportional, integral and derivative terms
KP := KPID[0]
KI := KPID[1]
KD := KPID[2]
//Call PID function
pid_direct(pid_period, Target_T, KP, KI, KD, dev, 1, 1)
```



Finally, at the end, all extended features should be disabled and the device should be closed.

```
ljdevice_close(dev)
```

The code for the pid_direct() function is given in the next segment. The next part of the code is used to read the temperature and to communicate with the PID controller continuously.

```
pid_direct(period, set_point, kp, ki, kd, ljdev, over, under)
{
  point_value := 0
  dt := 0
  stop := false
  error := 0
  preverror := 0
  inte := 0
  manipulatedValue := 0
  Temp := 0
  tgr := vector_create(2, true, 0)
  de := 0
  curr_time := timenow()
  tstart := curr_time
  config_a := 0
  roll_value := 80000
  //PID loop
  while(!stop)
  {
    Temp = timenow()
    dt = Temp - curr_time
    if(dt == 0)
      dt = 0.001
    curr_time = Temp
    point_value = ljdevice_read(ljdev, "AIN2_EF_READ_A") //read T
    Ta = point_value
    if ((Ta > (set_point + over)) || (Ta < (set_point -under)))
      stop = true
    tgr[0] = timenow() - tstart
    tgr[1] = Ta

    graph = join_mat_rows(graph, tgr)
    error = set_point - point_value
    de = error - preverror
    preverror = error
    inte = inte + dt * error
    manipulatedValue = kp * error + ki * inte + kd * de / dt
    if(manipulatedValue < 0)
    {
      manipulatedValue = 0
    }
    else if(manipulatedValue > 100)
    {
      manipulatedValue = 100
    }
    config_a = roll_value * manipulatedValue / 100
    ljdevice_write(dev,"DIO0_EF_CONFIG_A", config_a) //set control
    sleep(period)
  }
}
```

Here is the code for the auto-tunning function .

```
// pid autotune
pid_autotune_direct(period, set_point, ttype, ljdev, over, under)
{
  stop := false
  kp := 0
  ki := 0
  kd := 0
  cycles := 5
  finished := false
  loopinterval := 0
  pointvalue := 0
  Temp := 0
  mx := -1000000
  mn := 1000000
  thigh := 0
  tlow := 0
  minOutput := 0
  maxOutput := 100
  outputValue := maxOutput
  kpConstant := 0
  tiConstant := 0
  tdConstant := 0
  isModeP := 0
  // znModeP=0
  // znModePI=1
  // znModeBasicPID=2
  // znModeLessOvershoot=3
  // znModeNoOvershoot=4
  // znModePessenIntegral=5
  if(ttype == 0)
  {
      kpConstant = 0.5
      tiConstant = 1.0
      tdConstant = 0.0
      isModeP = 0.0
  }
```

```

else if(ttype == 1)
{
  kpConstant = 1/2.2
    tiConstant = 1/1.2
    tdConstant = 0.0
    isModeP = 1.0
}
else if(ttype == 2)
{
  kpConstant = 0.6
  tiConstant = 0.5
  tdConstant = 0.125
  isModeP = 1.0
}
else if(ttype == 3)
{
  kpConstant = 0.3
  tiConstant = 0.5
  tdConstant = 0.33
  isModeP = 1.0
}
else if(ttype == 4)
{
  kpConstant = 0.2
  tiConstant = 0.5
  tdConstant = 0.33
  isModeP = 1.0
}
else if(ttype == 5)
{
  kpConstant = 0.7
  tiConstant = 0.4
  tdConstant = 0.15
  isModeP = 1.0
}
ku := 0 // ultimate gain
tu := 0 //period of oscilations
counter := 0 //count cycles
paverage := 0
iaverage := 0
daverage := 0
output := true
config_a := 0
roll_value := 80000 /////
tutimer := timenow()
timer := tutimer
starttime := tutimer
tgr := vector_create(2, true, 0)
// tunning loop
while(!stop && !finished)
{
  finished = (counter >= cycles)
  pointvalue =ljdevice_read(ljdev, "AIN2_EF_READ_A") /////////////
  Tk := pointvalue
  ind :=((Tk > (set_point + over)) || (Tk < (set_point -under)))
  if ((counter >= 1) && ind)
   stop = true
  Temp = timenow()
  loopinterval = Temp - timer
  timer = Temp
```

```

    tgr[0] = Temp - starttime
    tgr[1] = pointvalue
    graph1 = join_mat_rows(graph1, tgr)
    mx = max(mx, pointvalue)
    mn = min(mn, pointvalue)
    if(output && (pointvalue > set_point))
    {
      output = false
      outputValue = minOutput
      Temp1 := timenow()
      thigh = Temp1 - tutimer
      if(thigh == 0)
        thigh = 0.001
      tutimer = Temp1
      mx = set_point
    }
    if(!output && (pointvalue < set_point))
    {
      output = true
      outputValue = maxOutput
      Temp2 := timenow()
      tlow = Temp2 - tutimer
      if(tlow == 0)
        tlow = 0.001
      tutimer = Temp2
      ku = (4 * (maxOutput - minOutput) / 2) / (cpi() * (mx -mn) / 2)
      tu = (tlow + thigh)
      print(tu)
      kp = kpConstant * ku
      ki = (loopinterval) * (kp * isModeP) / (tiConstant * tu)
      kd = (tdConstant * kp * tu) / (loopinterval)
      if(counter > 1)
      {
        paverage += kp
        iaverage += ki
        daverage += kd
      }
      mn = set_point
      counter += 1
      if(counter > cycles)
      {
        outputValue = minOutput
        kp = paverage / (counter -1)
        ki = iaverage / (counter -1)
        kd = daverage / (counter -1)
      }
    }
    config_a = roll_value * outputValue / 100
    ljdevice_write(dev,"DIO0_EF_CONFIG_A", config_a)
    sleep(period)
  }
  ret_val := vector_create(3, false, 0)
  ret_val[0] = kp
  ret_val[1] = ki
  ret_val[2] = kd
  return(ret_val)
}
```