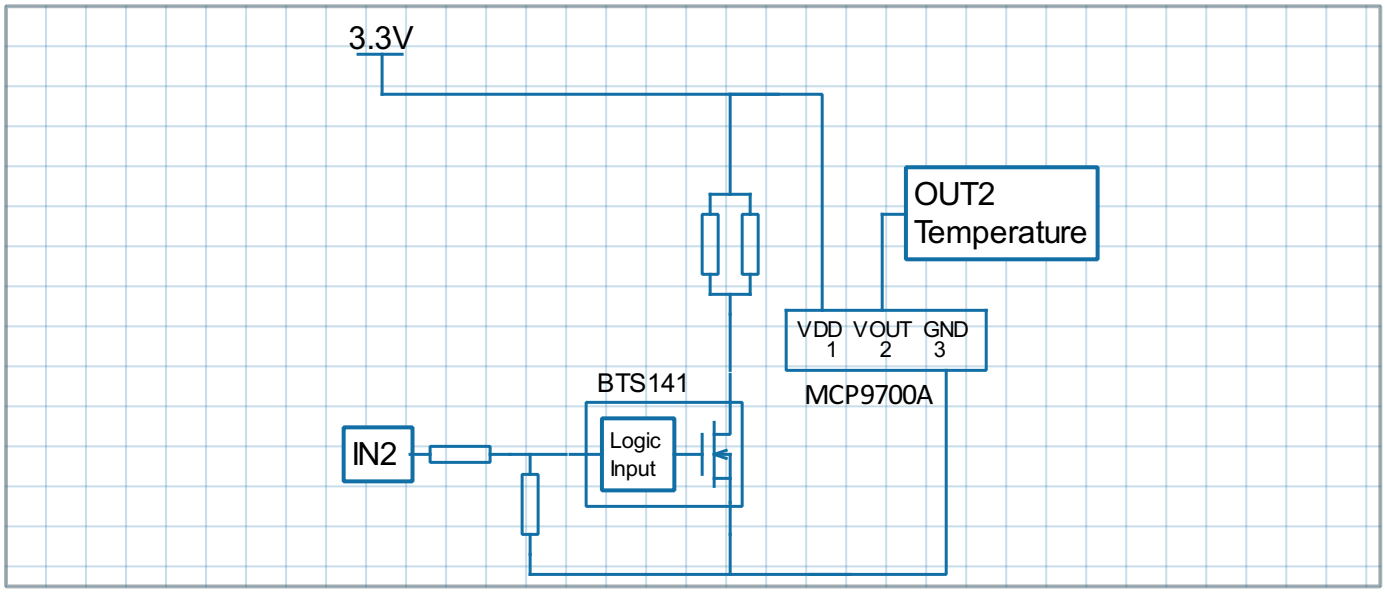
Temperature Control using LabJack and MatDeck's PID controller

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) and to measure the temperature using an analog input AIN. The control signal, PWM signal duty cycle, is set using a PID controller which is implemented in MatDeck.

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 for the PWM driver which heats up the resistor
- OUT2 - output from the temperature sensor in mV

Connection to ADC unit

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

Parts:

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

Using the LabJack T7 device in MatDeck

In this experiment, the LabJack T7 device is used to produce a digital PWM output which is connected to the IN2 signal. At the same time, the T7 is used to measure the temperature by collecting OUT2 signals 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.

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

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

Configuration of DIO EF PWM out

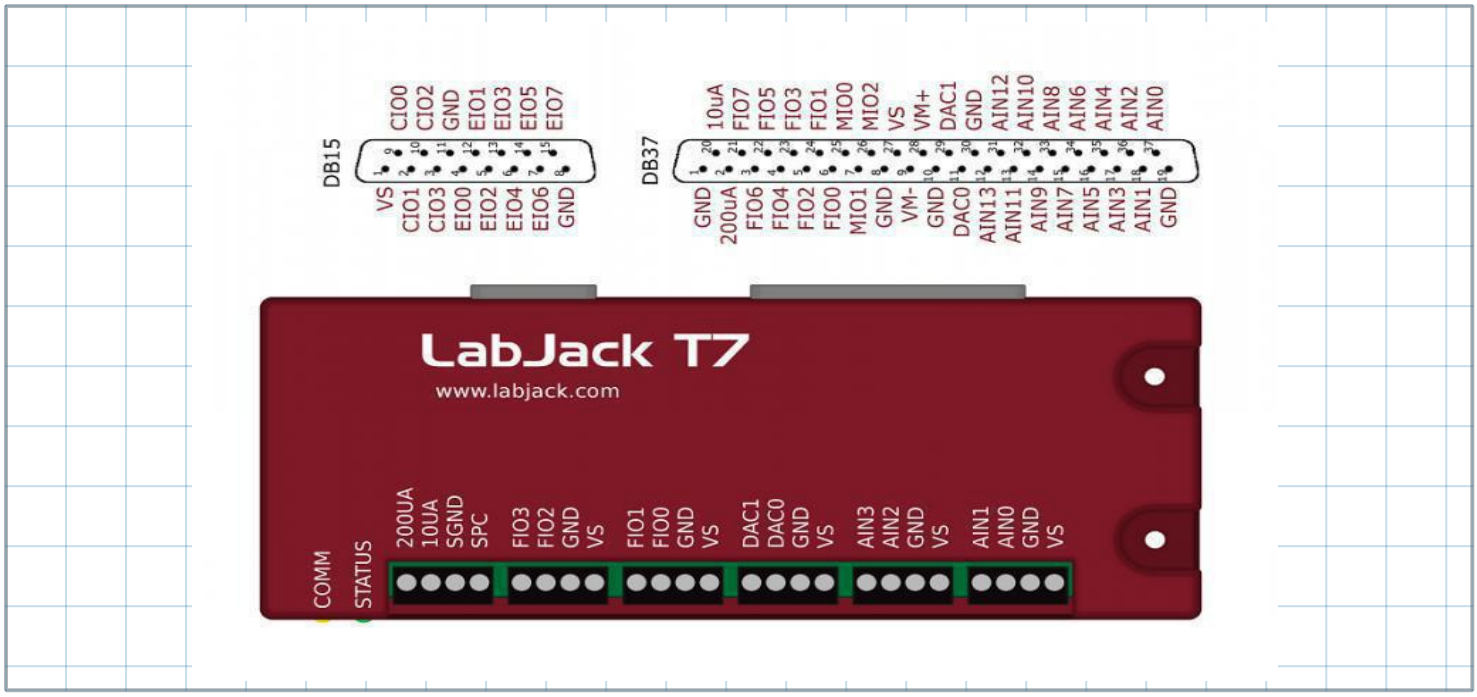
Here, DIO0 is used to produce the IN2 signals. When DIO0 is high, the transistor is switched on and the current through the resistors heats the temperature sensor. If DIO0 is low, the transistor is switched off and there is no current, thus the temperature will fall.

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, but 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. Before all the values are written into the appropriate registers, clock0 should be disabled.

```
2  //Disable clock0
3  ljdevice_write(dev,"DIO_EF_CLOCK0_ENABLE", 0)
4  //Setup Clock
5  ljdevice_write(dev,"DIO_EF_CLOCK0_DIVISOR", 1)//DIO_EF_CLOCK#_DIVISOR are
   1,2,4,8,16,32,64,or 256 pre-counter to devide 80MHz
6  ljdevice_write(dev,"DIO_EF_CLOCK0_ROLL_VALUE", 80000)//Devide 80MHz by
   8000 = 1000
```

PWM out at FIO0 (DIO0) is configured by setting DIO0_EF_INDEX equal to 0. A duty cycle is set by writing the appropriate value at DIO0_EF_CONFIG_A. If the desired value of the duty cycle is 50%, then DIO0_EF_CONFIG_A will be equal to half of the roll value, which is 40000.

```
7  //PWM Out at DIO0
8  ljdevice_write(dev,"DIO0_EF_ENABLE", 0)
9  //DIO Index valu 0 - PWM set
10 ljdevice_write(dev,"DIO0_EF_INDEX", 0)
11 //PWM duty cycle
12 ljdevice_write(dev,"DIO0_EF_CONFIG_A", 40000)
```



Enable Extended Features

Configured Extended Features should be enabled, this is done by writing one into the ENABLE register:

```
//Enable and run clock DIO18 and PWM DIO0
ljdevice_write(dev,"DIO_EF_CLOCK0_ENABLE", 1)
ljdevice_write(dev,"DIO0_EF_ENABLE", 1)
a := ljdevice_last_error("s") //Check for errors in configuration
```

```
a = "LJ_SUCCESS"
```

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.

```
ljdevice_write(dev, "AIN2_EF_INDEX", 1)
ljdevice_write(dev, "AIN2_EF_CONFIG_D", Slope)
ljdevice_write(dev, "AIN2_EF_CONFIG_E", Offset)
```

The temperature is automatically read using:

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

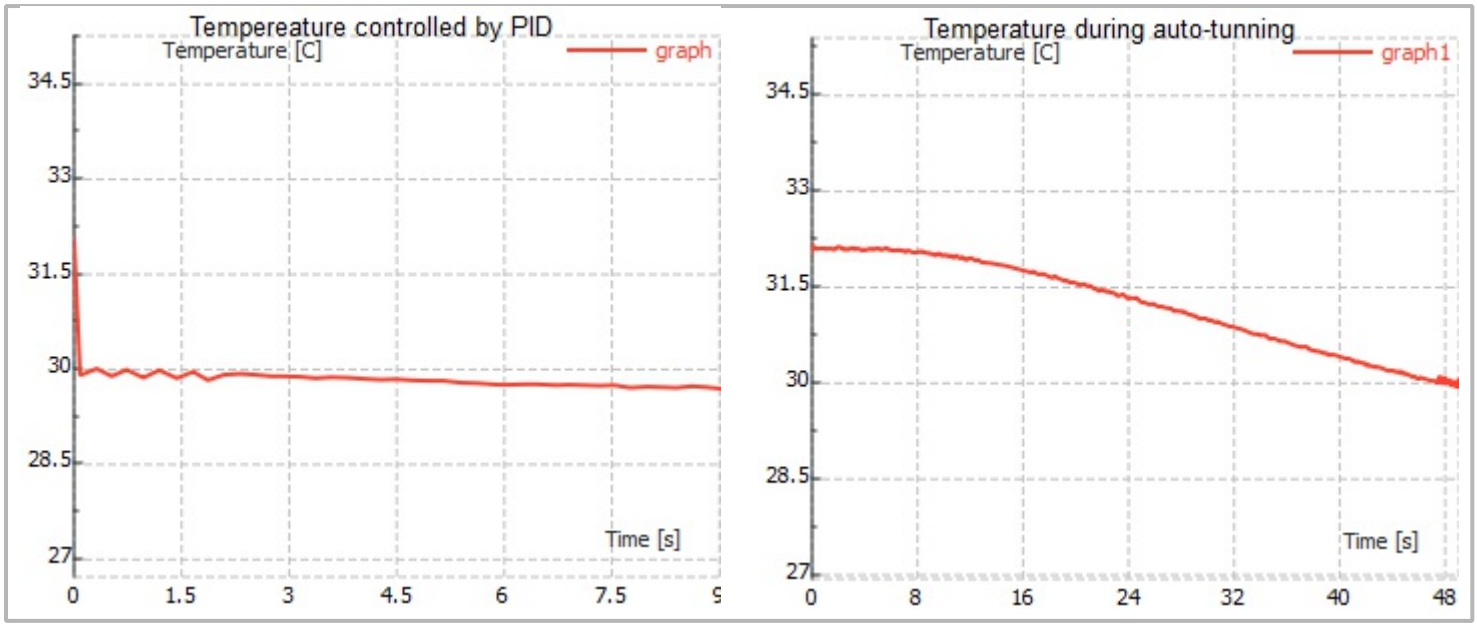
$$Ta = 29.696 \quad C$$

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. MatDeck's PID controller is used in real-time operation in the proposed heating 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, applies a correction based on 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 sets the 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 controller is used for the adjustment of 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, all extended features should be disabled and the device should be closed.

```
39 ljdevice_close(dev)
```

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

```
40 pid_direct(period, set_point, kp, ki, kd, ljdev, over, under)
41 {
42   point_value := 0
43   dt := 0
44   stop := false
45   error := 0
46   preverror := 0
47   inte := 0
48   manipulatedValue := 0
49   Temp := 0
50   tgr := vector_create(2, true, 0)
51   de := 0
52   curr_time := timenow()
53   tstart := curr_time
54   config_a := 0
55   roll_value := 80000
56   //PID loop
57   while(!stop)
58   {
59     Temp = timenow()
60     dt = Temp - curr_time
61     if(dt == 0)
62       dt = 0.001
63     curr_time = Temp
64     point_value = ljdevice_read(ljdev, "AIN2_EF_READ_A") //read T
65     Ta = point_value
66     if ((Ta > (set_point + over)) || (Ta < (set_point -under)))
67       stop = true
68     tgr[0] = timenow() - tstart
69     tgr[1] = Ta
70     graph = join_mat_rows(graph, tgr)
```

```
71
72      error = set_point - point_value
73      de = error - preverror
74      preverror = error
75      inte = inte + dt * error
76      manipulatedValue = kp * error + ki * inte + kd * de / dt
77      if(manipulatedValue < 0)
78      {
79        manipulatedValue = 0
80      }
81      else if(manipulatedValue > 100)
82      {
83        manipulatedValue = 100
84      }
85      config_a = roll_value * manipulatedValue / 100
86      ljdevice_write(dev,"DIO0_EF_CONFIG_A", config_a) //set control
87      sleep(period)
88    }
89  }
```

Here is the auto-tunning function code.

```
90  // pid autotune
91  pid_autotune_direct(period, set_point, ttype, ljdev, over, under)
92  {
93    stop := false
94    kp := 0
95    ki := 0
96    kd := 0
97    cycles := 5
98    finished := false
99    loopinterval := 0
100   pointvalue := 0
101   Temp := 0
102   mx := -1000000
103   mn := 1000000
104   thigh := 0
105   tlow := 0
106   minOutput := 0
107   maxOutput := 100
108   outputValue := maxOutput
109   kpConstant := 0
110   tiConstant := 0
111   tdConstant := 0
112   isModeP := 0
113   // znModeP=0
114   // znModePI=1
115   // znModeBasicPID=2
116   // znModeLessOvershoot=3
117   // znModeNoOvershoot=4
118   // znModePessenIntegral=5
119   if(ttype == 0)
120   {
121       kpConstant = 0.5
122       tiConstant = 1.0
123       tdConstant = 0.0
124       isModeP = 0.0
125   }
126   else if(ttype == 1)
127   {
```

```
128
129     kpConstant = 1/2.2
130       tiConstant = 1/1.2
131       tdConstant = 0.0
132       isModeP = 1.0
133   }
134   else if(ttype == 2)
135   {
136     kpConstant = 0.6
137     tiConstant = 0.5
138     tdConstant = 0.125
139     isModeP = 1.0
140   }
141   else if(ttype == 3)
142   {
143     kpConstant = 0.3
144     tiConstant = 0.5
145     tdConstant = 0.33
146     isModeP = 1.0
147   }
148   else if(ttype == 4)
149   {
150     kpConstant = 0.2
151     tiConstant = 0.5
152     tdConstant = 0.33
153     isModeP = 1.0
154   }
155   else if(ttype == 5)
156   {
157     kpConstant = 0.7
158     tiConstant = 0.4
159     tdConstant = 0.15
160     isModeP = 1.0
161   }
162   ku := 0 // ultimate gain
163   tu := 0 //period of oscilations
164   counter := 0 //count cycles
165   paverage := 0
166   iaverage := 0
167   daverage := 0
168   output := true
169   config_a := 0
170   roll_value := 80000 /////
171   tutimer := timenow()
172   timer := tutimer
173   starttime := tutimer
174   tgr := vector_create(2, true, 0)
175   // tunning loop
176   while(!stop && !finished)
177   {
178     finished = (counter >= cycles)
179     pointvalue =ljdevice_read(ljdev, "AIN2_EF_READ_A") ////////////
180     Tk := pointvalue
181     ind :=((Tk > (set_point + over)) || (Tk < (set_point -under)))
182     if ((counter >= 1) && ind)
183       stop = true
184     Temp = timenow()
185     loopinterval = Temp - timer
186     timer = Temp
187     tgr[0] = Temp - starttime
188     tgr[1] = pointvalue
```

```
189
190     graph1 = join_mat_rows(graph1, tgr)
191
192     mx = max(mx, pointvalue)
193     mn = min(mn, pointvalue)
194     if(output && (pointvalue > set_point))
195     {
196       output = false
197       outputValue = minOutput
198       Temp1 := timenow()
199       thigh = Temp1 - tutimer
200       if(thigh == 0)
201         thigh = 0.001
202       tutimer = Temp1
203       mx = set_point
204     }
205     if(!output && (pointvalue < set_point))
206     {
207       output = true
208       outputValue = maxOutput
209       Temp2 := timenow()
210       tlow = Temp2 - tutimer
211       if(tlow == 0)
212         tlow = 0.001
213       tutimer = Temp2
214       ku = (4 * (maxOutput - minOutput) / 2) / (cpi() * (mx -mn) / 2)
215       tu = (tlow + thigh)
216       kp = kpConstant * ku
217       ki = (loopinterval) * (kp * isModeP) / (tiConstant * tu)
218       kd = (tdConstant * kp * tu) / (loopinterval)
219       if(counter > 1)
220       {
221         paverage += kp
222         iaverage += ki
223         daverage += kd
224       }
225       mn = set_point
226       counter += 1
227       if(counter > cycles)
228       {
229         outputValue = minOutput
230         kp = paverage / (counter -1)
231         ki = iaverage / (counter -1)
232         kd = daverage / (counter -1)
233       }
234     }
235     config_a = roll_value * outputValue / 100
236     ljdevice_write(dev,"DIO0_EF_CONFIG_A", config_a)
237     sleep(period)
238   }
239   ret_val := vector_create(3, false, 0)
240   ret_val[0] = kp
241   ret_val[1] = ki
242   ret_val[2] = kd
243   print(kp)
244   print(ki)
245   print(kd)
246   return(ret_val)
247 }
```