Rapid Electronics Prototyping with Arduino

Arduino Projects: Practical Control Projects

By

Benjamin Tyler, PE
PDH4Engineers

April 13, 2023

Contents

Introduction

This course provides practical experience implementing commonly used controls with Arduino. Arduino is an ideal platform for rapid development of programmable electronics. Arduino has simplified the software and hardware aspects of electronics development. The Arduino ecosystem has lots of freely available and useful libraries that greatly simplify coding your projects. Additionally, there is a plethora of open, documented development boards that you can use as a guide for the hardware aspect of your projects. You leverage all the freely available resources as an engineer to rapidly develop a working solution.

In this course, we create a few practical control projects with Arduino. These projects are simplified versions of real world applications. We use Arduino as if it is a low cost, open source version of SCADA hardware, programmable logic controller, or PID controller. This might sound grandiose at first glance, but Arduino hardware is perfectly capable of many tasks. The projects in this course include a timer controller, a temperature limit controller, a state machine controller, and a PID (proportional-integral-derivative) controller.

The state machine and PID projects have an abundance of background theory, but this course focuses purely on implementation.

If you are not familiar with Arduino or have not done much programming before, I recommend you take the preceding Introduction to Arduino course first. The previous course covers topics like installing Arduino, installing libraries, digital and analog I/O, and serial communication.

Author Introduction

This course has been produced by Benjamin Tyler, PE, who has over 15 years designing embedded systems, data acquisition and automation systems, and mobile apps.

Note About Source Code In This Course

Source code for this course is under the “MIT No Attribution” license. Do not use this code for medical, aviation, or other safety-critical applications. You may use this code freely, even for commercial applications. License included below:

Suggested Course Materials

While this course can be completed without an Arduino board, you will get much more out of this course if you follow along with one. I recommend the Arduino Uno, as it is inexpensive and very capable. Furthermore, I recommend you buy a kit that contains an Arduino board, sensors, and other components. You can use an Arduino board other than the Uno, but you might need to make minor changes to the project code. The Arduino Leonardo board is a great alternative to the Uno, and one advantage it has over the Uno is that it can function as a USB peripheral. There are a few places you can buy an Arduino board or kit that includes the board:

- ***www.arduino.cc***
- ***www.amazon.com***
- ***www.sparkfun.com***
- ***www.adafruit.com***
- ***www.microcenter.com***

The following are Amazon affiliate links to suitable kits and parts:

Elego Uno Super Start Kit

Elegoo Uno Complete Starter Kit

Elegoo Complete Uno Starter Kit

Lafvin Super Starter Kit

Adafruit NeoPixel Strip

Adafruit Universal Thermocouple Amplifier MAX31856 Breakout

Make sure to get the following components, or a kit that contains them:

- Arduino Uno, Leonardo, Zero, Mega, Micro, Nano, or any other Arduino compatible board
- solderless breadboard
- Dupont jumper wires
- DS3231 RTC module
- 5V relay
- 2N2222 or similar NPN transistor
- 1N4007 or similar diode

- (2x) 1k Ohm resistors
- 220 Ohm resistor
- photoresistor or phototransistor
- LEDs
- MAX31856 breakout board from Adafruit
- K-type thermocouple

Project 1 – Timer

Parts needed for this project

- Arduino Uno or other Arduino board
- Solderless breadboard
- Jumper wires
- DS3231 RTC module
- 5V (input voltage) relay
- 2N2222 NPN transistor
- 1N4007 diode
- 2x 1k resistor
- LED

Project explanation

In this mini project we make a timer control. Timers are very useful for controlling simple processes like turning on lights at a certain time of the day. Like with many commercial timers, our timer's output is a normally open SPST relay. Relays are electromechanical switches that allow a microcontroller to switch much more current than a microcontroller can source or sink.

The relay's specs determine what kinds of loads it can drive. In the case of the relay used in this mini project, the relay is rated up to 40V at 10A DC and 120VAC or 220VAC at 10A.

There are two issues with the relay. First, the input coil requires more current to energize than the Arduino I/O pin can source or sink. Second, The relay is an inductive load, so it will create voltage spikes when current to it is cut off (due to Faraday's law of induction). These issue about supplying enough current to the input coil can be solved by using a transistor to drive the relay's input coil. We use a 2N2222, but any similar NPN transistor will work. The voltage spike issue is solved by putting a flyback (also known by freewheeling and other names) diode across the relay. This creates a return path for voltage surges.

The DS3231 real-time clock has two programmable alarms. They can be set to run every minute, every hour, every day, or at specific dates.

Build the circuit below.

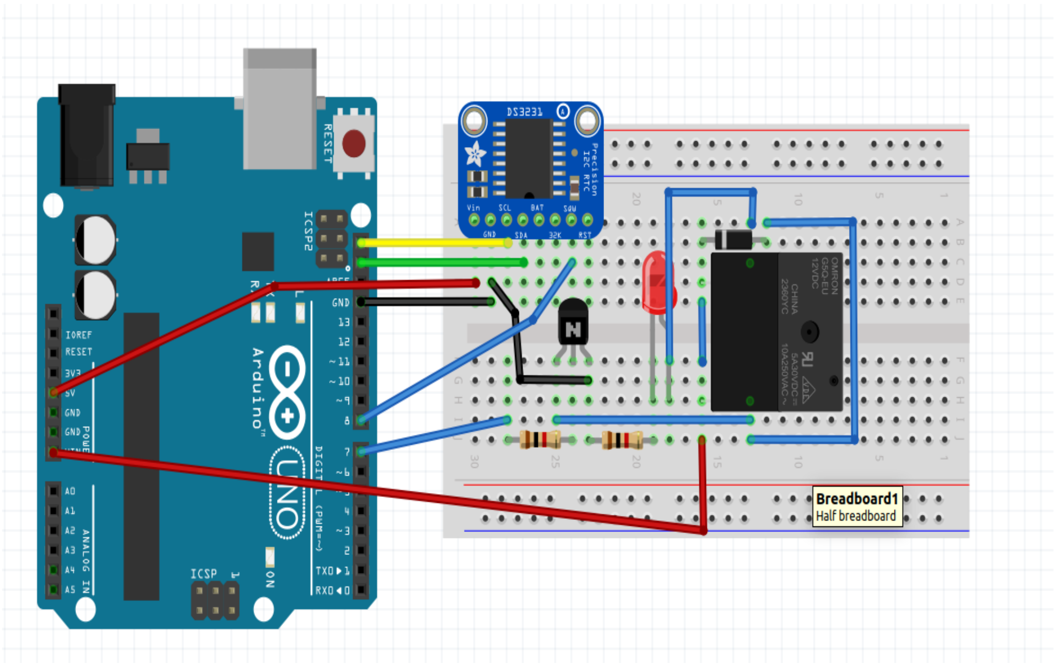

Figure 1: Project 1 Circuit

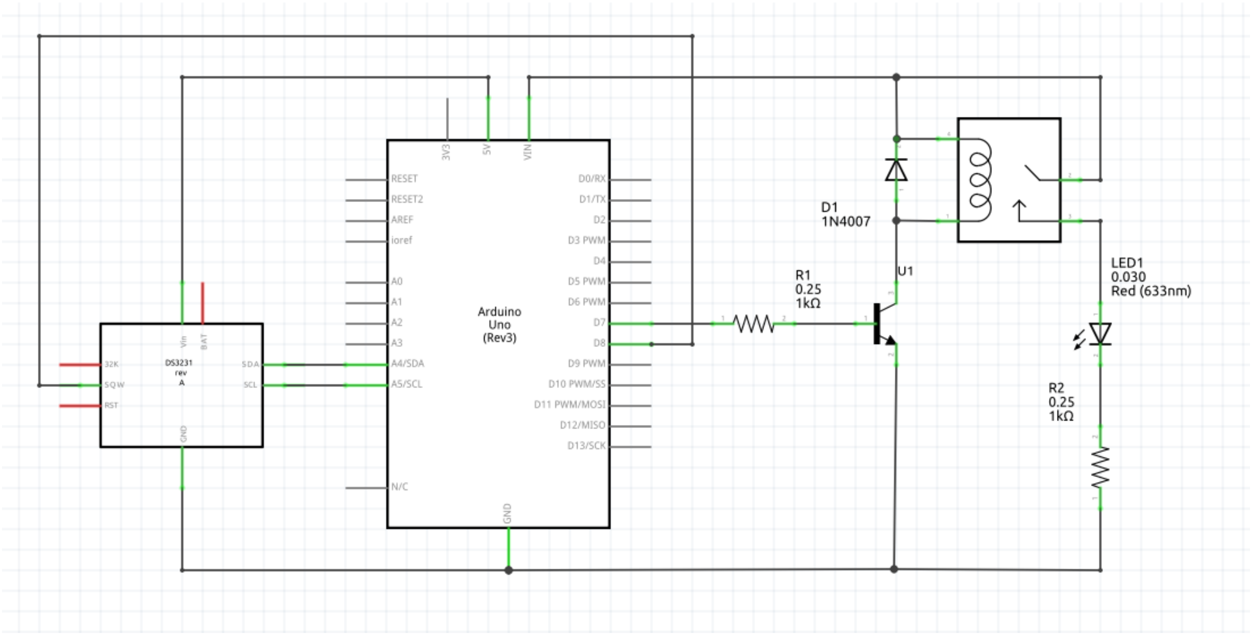

Figure 2: Project 1 Schematic

In the Library Manager, install RTCLib from Adafruit. Some of the classes and functions we will use from the RTCLib include:

RTC_DS3231() – Constructor for a DS3231 object.

.begin (TwoWire *wireInstance=&Wire) – Starts I2C communication and returns true if successful.

.adjust (const DateTime &dt) – Sets the date and time for the DS3231.

.now() - Returns a DateTime object with the current date/time.

DateTime (uint16_t year, uint8_t month, uint8_t day, uint8_t hour=0, uint8_t min=0, uint8_t sec=0) – Constructor for a DateTime object.

.setAlarm1(const DateTime &dt, Ds3231Alarm1Mode alarm_mode) – Sets when Alarm 1 fires.

Project specifications

1. Set alarm 1 to run 5 seconds after power on/reset for 2 seconds.
2. Set alarm 2 to run every minute for 5 seconds.
3. When an alarm fires, we determine whether alarm 1 or alarm 2 fired.
4. In alarm handler, enable the relay output.
5. In the main loop, check how long the relay has been enabled and turn it off after being on for 5s.

Project code

Now let’s make the project. Type or copy/paste the following code into Arduino IDE.

```
#include <RTClib.h>

#define ALARM_PIN 2 // connected to SQW

#define RELAY_PIN 7
// 1
RTC_DS3231 rtc;
DateTime relayOnTime;
bool relayEnabled = false;
bool alarmFired = false;

void setup() {
      Serial.begin(9600);
      pinMode(RELAY_PIN, OUTPUT);
      digitalWrite(RELAY_PIN, LOW);
      pinMode(ALARM_PIN, INPUT_PULLUP);
      attachInterrupt(digitalPinToInterrupt(ALARM_PIN), onAlarm, FALLING);
// 2
      if(!rtc.begin()) {
            Serial.println("RTC not detected");
            Serial.flush();
            while (1) delay(10);
      }

      if (rtc.lostPower()) {
            rtc.adjust(DateTime(F(__DATE__), F(__TIME__))); // sets time
to when code was compiled
      }
      rtc.disable32K();
      rtc.clearAlarm(1);
      rtc.clearAlarm(2);
      rtc.writeSqwPinMode(DS3231_OFF);
// 3
      if(!rtc.setAlarm1(
            rtc.now() + TimeSpan(5),
            DS3231_A1_Date
      )) {
            Serial.println("Error: Alarm1 not set");
      }else {
            Serial.println("Alarm1 set");
      }

      if(!rtc.setAlarm2(
            rtc.now(),
            DS3231_A2_PerMinute
            )) {
            Serial.println("Error: Alarm2 not set");
      }else {
            Serial.println("Alarm2 set");
      }
}
// 4
void loop() {
      char date[10] = "hh:mm:ss";
      rtc.now().toString(date);
      Serial.println(date);

      if (alarmFired) {
            alarmFired = false;
            if (rtc.alarmFired(1)) {
                  rtc.clearAlarm(1);
                  rtc.disableAlarm(1);
                  Serial.println("Alarm1 fired");
                  digitalWrite(RELAY_PIN, HIGH);
                  relayEnabled = true;
                  relayOnTime = rtc.now() + TimeSpan(2);
            }
            if (rtc.alarmFired(2)) {
                  rtc.clearAlarm(2);
                  Serial.println("Alarm2 fired");
                  digitalWrite(RELAY_PIN, HIGH);
                  relayEnabled = true;
                  relayOnTime = rtc.now() + TimeSpan(5);
            }
      }
// 5
      if (relayEnabled && rtc.now() > relayOnTime) {
            digitalWrite(RELAY_PIN, LOW);
            relayEnabled = false;
      }

      delay(1000);
}
// 6
void onAlarm() {
      Serial.println("fired");
      alarmFired = true;
}
```

Code explanation

1. At the beginning of the program we assign pin numbers, create a DS3231 object, and create flags that indicate if an alarm has fired and if the relay is on.

2. This setup code sets digital pins for reading the alarm interrupt and turning on the relay. The DS3231 object, rtc, is started. Then the time is set if a power disruption has been detected. The time is gotten by using a macro that grabs the host computers system time when the code is compiled. Then there is code that sets up the DS3231. Alarms are cleared and functionality that isn't used is disabled.

3. Then the alarms are set. Alarm1 is set to fire at the current time plus 5 seconds. Since the DS3231_A1_Date is set, the alarm will fire at that date, so alarm1 will not repeat. On the other hand, alarm2 is set to fire every minute when the value for seconds is 0.

4. In the main loop, the current time is printed at the beginning of the loop. Then if the alarmFired flag is true, then alarm1 and alarm2 are checked to see which one fired. If an alarm is detected, then the alarm is cleared, the relay output pin is set to high to turn on the relay, and the relay on-time is set to the current time plus the desired duration. The relayEnabled flag is set to true. The relay stays on from this point onwards.

5. Then the code checks to see if the relayEnabled flag is set and if current time has passed the on time duration. If yes, then that means the relay has been on for the desired duration of time. The relay is turned off, and the relayEnabled flag is reset.

6. This is the interrupt handler that fires when the alarm interrupt pin fires. The handler sets a flag that is handled in the main loop.

Project Review

So far we have used alarms that fire once and every minute. The alarms can also be set for days of the week, or, with some additional logic, for any trivial point of time. To see the documentation for the Adafruit DS3231 driver, visit ***https://adafruit.github.io/RTClib/html/index.html***.

Now that we have implemented a timer that is useful for many applications, let’s change it a little in the next project.

Project 2 – Temperature Limit Controller

Parts needed for this project

- Arduino Uno or other Arduino board
- Solderless breadboard
- Jumper wires
- 5V (input voltage) relay
- 2N2222 NPN transistor
- 1N4007 diode
- 2x 1k resistor
- LED
- MAX31856 breakout board from Adafruit
- K-type thermocouple

Project explanation

In this mini project we build a limit controller. This is a very common type of controller that turns on or turns off a load when the temperature reaches a specified temperature. In typical applications, the thermal limit can be well above 100° C, and thermocouples are a common sensor for a wide range of temperatures. So, we use a MAX31856 thermocouple amplifier to read a K-type thermocouple which is suitable for many applications.

This circuit is almost the same as the last, except we substitute the MAX31856 breakout board for the DS3231. I recommend the Adafruit MAX31856 board, but a board from Sparkfun or anywhere else will do.

One difference between the MAX31856 and the DS3231 is that the MAX chip communicates over SPI instead of I2C. SPI is a serial, synchronous interface, meaning it needs a clock signal that is synchronized between master and slave devices in order to transfer data. SPI devices do not need to decode addresses, because chip select lines determine which chip is listening.

Build the following circuit:

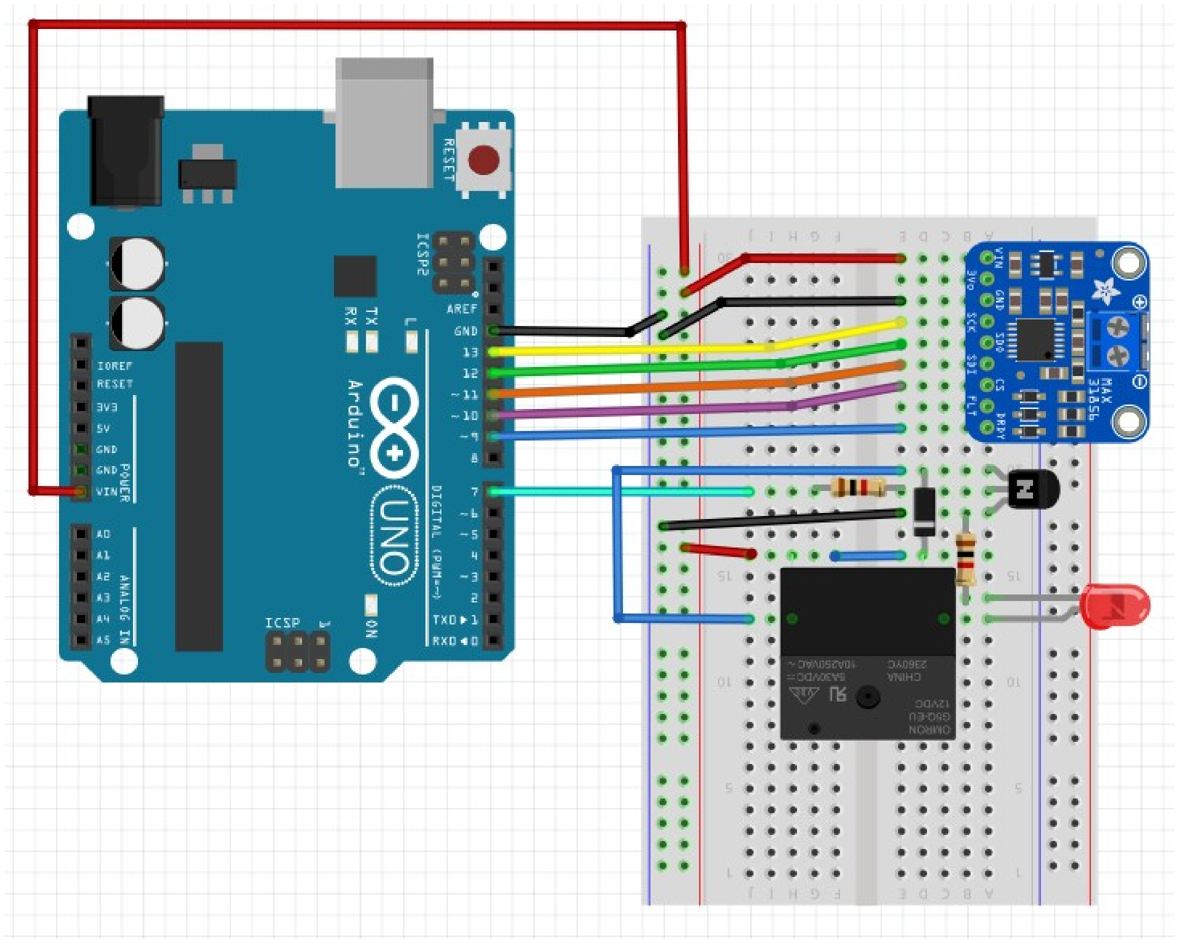

Figure 3: Project 2 Circuit

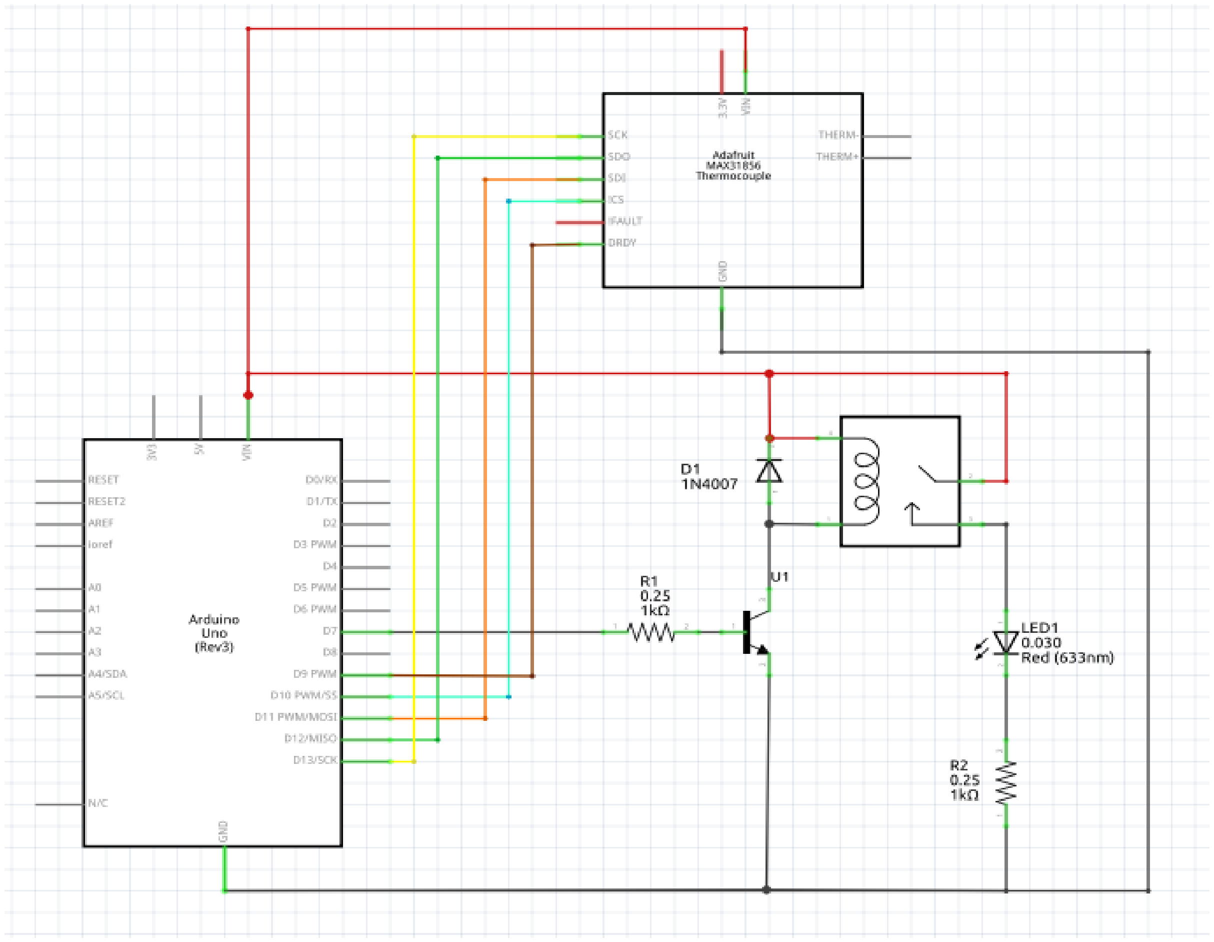

Figure 4: Project 2 Schematic

Next install the Adafruit MAX31856 library with the Library Manager. Some classes and functions we will use include:

Adafruit_MAX31856(CS, DI, DO, CLK) – This takes in the SPI port pins and constructs a MAX3156 object.
.begin() - Initializes the sensor chip.
.setThermoCoupleType(MAX31856_TCTYPE_K) – Sets the thermocouple to K type.
.setConversionMode(MAX31856_CONTINUOUS) – Makes the chip continuously read the thermocouple. A one-shot option is also available.
.readThermocoupleTemperature() - Returns the temperature in Celsius as a float. For simplicity, we convert this value to an int.
.readCJTemperature() - Returns the cold junction temperature, the internal reference for the thermocouple.
.readFault() - Returns a one byte fault flag. We will not use this in the example, but it is recommended to use it in production. Connect the MAX31856's FLT pin to an I/O pin, and use readFault when the pin goes low.

The header file has all the settings and functions you can use with the MAX31856:
https://github.com/adafruit/Adafruit_MAX31856/blob/master/Adafruit_MAX31856.h

Project specifications

1. Read the thermocouple temperature continuously.
2. Keep relay closed. In our case, the relay is normally open, so the relay needs to stay energized.
3. If the temperature reaches a given threshold, in our case 80° C, open the relay.
4. After the threshold is met, blink the built-in LED until the board is reset. (Reset by pressing the reset button.)

Project code

Next type or copy/paste the following code into Arduino IDE:

```
// 1
#include <Adafruit_MAX31856.h>
#define DRDY_PIN 9 // pin goes low when converted temperature ready
#define RELAY_PIN 7

Adafruit_MAX31856 tc = Adafruit_MAX31856(10);

int shutoffTemp = 80; // deg C, relay opens at this temperature

void setup() {
      Serial.begin(9600);
      pinMode(RELAY_PIN, OUTPUT);
      digitalWrite(RELAY_PIN, HIGH);
// 2
      pinMode(DRDY_PIN, INPUT);
      pinMode(LED_BUILTIN, OUTPUT);
      digitalWrite(LED_BUILTIN, LOW);

      if (!tc.begin()) {
            Serial.println("Error initializing thermocouple");
            while (1) delay(10);
      }

      tc.setThermocoupleType(MAX31856_TCTYPE_K);
      tc.setConversionMode(MAX31856_CONTINUOUS);
}

void loop() {
	int idle = 0;
// 3
	while (digitalRead(DRDY_PIN)) {
		if (idle++ > 200) {
			idle = 0;
		}
	}

	int reading = int(tc.readThermocoupleTemperature());

	Serial.print("Thermocouple reads: ");
	Serial.println(reading);
// 4
	if (reading >= shutoffTemp) {
		digitalWrite(RELAY_PIN, LOW);
		Serial.println("Temperature has exceeded threshold, powering 
down load.");
		Serial.println("Press reset button on board to power on 
load.");

		while (1){
			digitalWrite(LED_BUILTIN, HIGH);
			delay(1000);
			digitalWrite(LED_BUILTIN, LOW);
			delay(1000);
		}
	}
}
```

Code explanation

1. First we assign the pins that indicate when a temperature reading is ready and toggle the relay. We create a Adafruit_MAX31856 object that has all the thermocouple amplifier functionality. Then we establish a temperature threshold for shutting off the relay.

2. In setup, we configure the RELAY_PIN as an output and the DRDY_PIN as an input. Then we configure the MAX31856 to read continuously and to use a K type thermocouple.

3. Here the code polls the DRDY_PIN until it goes low. Once the pin goes low, the thermocouple is read.

4. If the temperature reading equals or is greater than the shutoff threshold, then the relay is turned off. The code goes into a loop blinking the built-in LED until the board is reset.

Project review

In this project we altered the first project to act as a limit controller. This is our first project to use SPI, serial peripheral interface, to communicate with a device. Since we used the Adafruit driver for the MAX31856, all the low level details have been abstracted away. This highlights one of the advantages of Arduino, that so much functionality is available off the shelf to let you rapidly create projects.

Project 3 – State Machine Controller

Parts needed for this project

- Arduino Uno or other Arduino board
- 2x Tactile switch (momentary push button)
- Solderless breadboard
- Jumper wires

Project explanation

A finite state machine (FSM) is an abstract machine that has a limited set of states and can only be in one state at a time. While a state machine can be used to model and implement complex behavior, it cannot implement any trivial algorithm as a Turing machine can. Each state, or node in a state machine diagram, can have an output value and internal state information. A state machine transitions from one state to another based on a trigger, which can be an input, a timer, or some other combination of conditions. Transitions and states can have actions associated with them.

A classic example of a state machine is a traffic light. The lights at an intersection rotate through a pattern of green, yellow, red, green arrows, blinking yellow arrows, crosswalk lights, etc. The lights may change state based on a timer, a sensor detecting a waiting car, or the press of a crosswalk button. A simplified state diagram for an intersection with one light with no button for crosswalk might look like this:

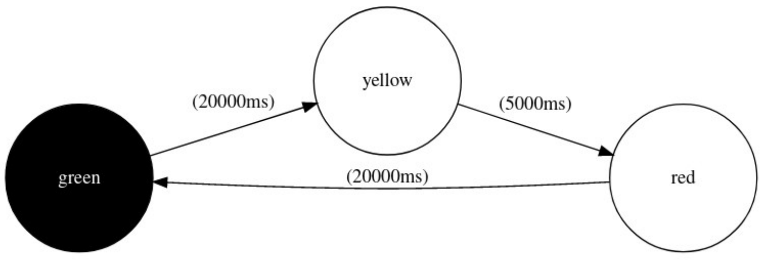

Figure 5: Simple state diagram of traffic light. Made with gravizo.com

In this example state diagram, the traffic light transitions from one color to another after a set delay. The initial state is indicated by the solid black fill. The green and red lights last for 20 seconds, and the yellow light lasts for 5 seconds.

Let's examine another simple example, a light switch. The switch is a single pole rocker switch. There are only two states, On and Off.

Figure 6: Run-of-the-mill light switch

There are two actions that you can take with this type of switch. Press the top On position or push the bottom Off position. If the light switch is in the On position, and the Off position is pushed, the light is turned off. If the light switch is in the On position, but the On position is pressed, nothing happens. The state diagram for this looks like this:

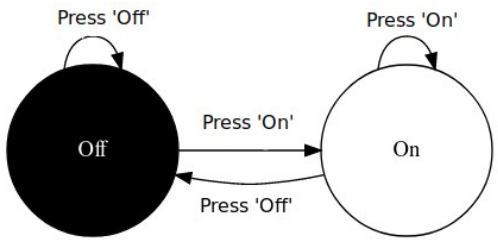

Figure 7: State diagram for rocker type of light switch. Made with graviso.com

State machines are great for modeling systems, and Arduino has libraries that simplifies implementing state machines in code. Using a library helps prevent your state machine from becoming a mess of nested for loops, while loops, switch statements, and delays.

In this project, we are going to model the operation of a toaster.

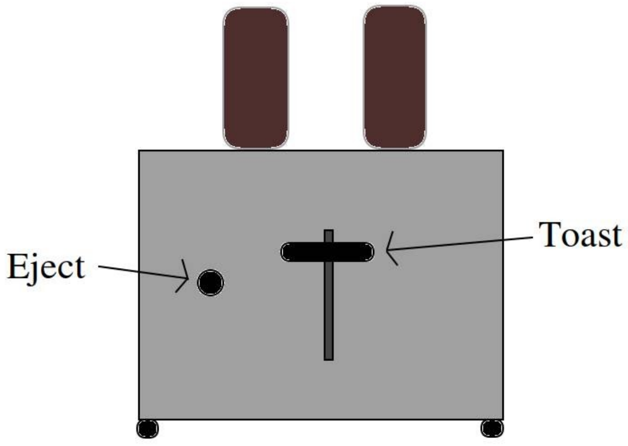

Figure 8: A crude depiction of a toaster with buttons labeled and bread loaded.

Please build the following circuit:

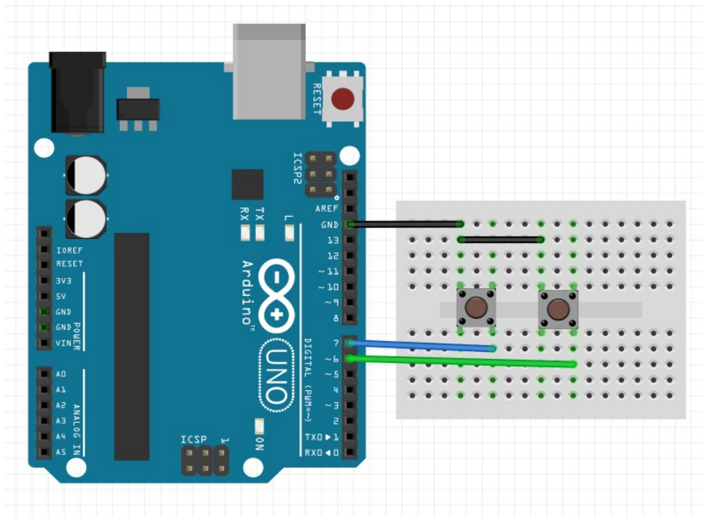

Figure 9: Circuit for state machine project

Install the SimpleFSM library written by Lennart Hennigs. Install the ezButton as well if it isn't already installed. There are three main kinds of objects in the SimpleFSM library: the state machine, states, and transitions. Some classes and functions we will use include:

SimpleFSM fsmObject - Constructs a new state machine object.

.add(Transition t[] | TimedTransition t[], int size) – Adds an array of states or timed states. Needs the length of the array.

.setInitialState(State* state) – Sets the initial state.

.setFinishedHandler(CallbackFunction f) – Sets a function that runs after changing states.

.setTransitionHandler(CallbackFunction f) – Sets a function that runs after a transition finishes.

.trigger(int event_id) - Sets a trigger, which in turn can cause a transition.

.run(int interval = 1000, CallbackFunction tick_cb = NULL) – Starts the state machine and sets how often the state machine updates.

.reset() - Returns station machine to initial state.

.getState() => State* - Returns a pointer to the current state.

State(String name, CallbackFunction on_enter = NULL, CallbackFunction on_state = NULL, CallbackFunction on_exit = NULL, bool is_final = false) – Constructs a State object. Each state needs a name and can have functions associated with entering and exiting the state, as well as when they are in a state. The on_state function, if defined, runs every every loop of the state machine as defined by state machine's interval.

Transition(State* from, State* to, int event_id, CallbackFunction on_run = NULL, String name = "", GuardCondition guard = NULL) – Creates a Transition object. The transition maps one state to another and is associated with a trigger event. The transition can be blocked with a a guard condition.

TimedTransition(State* from, State* to, int interval, CallbackFunction on_run = NULL, String name = "", GuardCondition guard = NULL) – Creates a TimedTransition object. It is similar to a normal transition except that instead of being triggered by an event, it is triggered by a time delay. The interval is in milliseconds.

Project specifications

1. We are modeling the operation of a toaster. There are two inputs, a button that initiates toasting and a button that ejects the toast before the timer expires. The output is a LED that represents the toaster's heating element.
2. Pushing down on the toast button lowers the bread into the toaster, turns on the heating element, starts a timer, and latches the button in the ON position. For demonstration purposes, we will use a momentary tactile switch (push button) that is common in kits. To simulate latching, we'll use the ezButton library. The ezButton library was covered in the last Arduino course about interfacing common hardware.

3. When the timer expires and the toast button is still latched ON, the toast button releases up to the OFF position, the heating element turns off, and the toast raises.
4. If the eject button is pressed while toasting, it releases the toast button, turns off the heating element, and resets the timer. The eject button is momentary, normally off.
5. If the eject button is pressed while not toasting, nothing happens.
6. Pressing both buttons at the same time has no effect.

Project code

Type or copy/paste the following code into Arduino IDE. Open the Serial Monitor to the serial messages.

```
#include <SimpleFSM.h>
#include <ezButton.h>
#include <avr/wdt.h>

// 1
#define HEAT_ELEMENT LED_BUILTIN
#define TOAST_DELAY 10000 // 10 seconds for demo purposes

ezButton toast_btn(6);
ezButton eject_btn(7);
SimpleFSM toaster;

// 2
void heat_on() {
	Serial.println("Entering State: ON");
	digitalWrite(HEAT_ELEMENT, HIGH);
}

void heat_off() {
	Serial.println("Entering State: OFF");
	digitalWrite(HEAT_ELEMENT, LOW);
}

State s0 = State("off", heat_off, NULL, NULL);
State s1 = State("on", heat_on, NULL, NULL);

// 3
void on_to_off() {
	Serial.println("Transition: ON -> OFF");
	wdt_enable(WDTO_120MS); // kind of a hack
}

void off_to_on() {
	Serial.println("Transition: OFF -> ON");
}

void no_change() {
	Serial.println("Transition: No change");
}

enum triggers {
	toast_button_pressed,
	eject_button_pressed
};

Transition transitions[] = {
	Transition(&s0, &s1, toast_button_pressed, off_to_on),
	Transition(&s1, &s0, eject_button_pressed, on_to_off),
	Transition(&s0, &s0, eject_button_pressed, no_change),
	Transition(&s1, &s1, toast_button_pressed, no_change)
};

// 4
void timed_on_to_off() {
	Serial.println("Timed Transition: ON -> OFF");
}

bool guard() {
	Serial.println("Guard function runs");
	return true;
}

TimedTransition timed_transitions[] = { TimedTransition(&s1, &s0, TOAST_DELAY,
timed_on_to_off, "", guard) };

int num_transitions = 4; // sizeof(transitions) / sizeof(Transition)
int num_timed_transitions = 1;

// 5
void setup() {
	Serial.begin(9600);
	pinMode(HEAT_ELEMENT, OUTPUT);

	toast_btn.setDebounceTime(100);
	eject_btn.setDebounceTime(100);

	toaster.add(transitions, num_transitions);
```

```
        toaster.add(timed_transitions, num_timed_transitions);
        toaster.setInitialState(&s0);
}

// 6
void loop() {
        toaster.run();
        toast_btn.loop();
        eject_btn.loop();

        if (toast_btn.isPressed() && eject_btn.isPressed()) {
                Serial.println("Both buttons pressed");
                return;
        }

        if (toast_btn.isPressed()) {
                Serial.println("Toast button pressed");
                toaster.trigger(toast_button_pressed);
        }

        if (eject_btn.isPressed()) {
                Serial.println("Eject button pressed");
                toaster.trigger(eject_button_pressed);
        }
}
```

Code explanation

1. First, create an alias for the Arduino board's built-in LED, and create a constant for the duration of time the "toaster" stays on. Create objects for the Toast button on pin 6 and the Eject button on pin 7. Create a SimpleFSM object.

2. In this code segment we create two states, s0 and s1. s0 is the "On" state, and s1 is the "Off" state. Each state has a function that fires once the state machine enters that state. heat_on fires when the state machine enters the "On" state, and heat_off fires in the "Off" state.

3. This code segment defines the transitions between states. Transitions from states to themselves need to be defined too if they are possible given the inputs- ie pressing the "On" button when the state machine is in the "On" state. If a state machine is missing a transition that it needs to take, the code can go off into the weeds. Each transition has an associated function that fires when the transition is used.

We have one issue here. In the scenario where the Toast button is pressed and the toaster is toasting. If the Eject button is pressed, then the state machine needs to transition to the "Off" state and cancel the timed transition that would fire if not for pressing Eject. Unfortunately the SimpleFSM library doesn't provide a way to reset the timer for the TimedTransition. So, even if we press Eject to go from "On" to "Off" before the timer expires, then the TimedTransition will fire immediately the next time the Toast button is pressed. Since we cannot reset the TimedTransition's timer value, we use a watchdog timer to reset the Arduino board. The watchdog resets the board after waiting a nominal 120 milliseconds, and the Arduino returns to its initial state.

4. The TimedTransition is defined here. The function associated with the TimedTransition is defined as well as a guard function. The guard can block the TimedTransition from running until it returns true. The guard function here is just for demonstration purposes, as it has no useful logic. The transitions are defined in an array, since the state machine object needs them in an array. The state machine needs to know how many transitions are in the array. The number of elements can be calculated automatically as in the comment next to num_transitions or manually as done in our code.

5. In setup we configure the serial port and the I/O pin for the imaginary heating element. Then we set the debounce values for each button to 100 ms. We add the transitions and timed transitions to the state machine and set the initial state.

6. In the run loop we update the state machine and the buttons. Then we check for button presses. If both buttons are pressed at the same time, nothing happens. If either of the buttons are pressed individually, a button pressed event is triggered.

Project review

We have successfully implemented a state machine on Arduino. As you can see, using a state machine is very useful for modeling and implementing systems that can be broken up into clearly defined states. The SimpleFSM library for Arduino has a lot of useful functionality that allows you to make complex state machines with a minimum of pain. See the Github page for more information: **https://github.com/LennartHennigs/SimpleFSM**.

Project 4 – PID Controller

Parts needed for this project

- Arduino Uno or other Arduino board
- Solderless breadboard
- Jumper wires
- white LED
- 220 Ohm resistor
- 1k Ohm resistor
- photocell (photoresistor) or phototransistor

Project explanation

The PID (proportional, integral, derivative) controller is commonly used in control systems and uses a feedback loop to reach the desired setpoint. It is often used to regulate speed, temperature, flow, and other process variables. It works by calculating the difference between the system’s setpoint and the measured process variable to get an error value. It calculates a correction based on the proportional, integral, and derivative gains to apply to the control variable. The control variable determines to what degree an actuator acts on the process under control. The PID adjusts the control variable to minimize error, making the process converge to the setpoint.

PID works best on processes that are repeatable, ie an input to a system has a similar output each time the PID controller runs.

Let’s quickly clarify the terms used with PID controls. The terms are often expressed as constants or as continuous time variables. The Arduino library we are using has a slightly different nomenclature we need to map to the standard definitions.

setpoint: SP or $r(t)$

process variable: PV or $y(t)$ or Input

control variable: $u(t)$ or Output

error: $e(t) = r(t) – y(t) = SP - PV$

A classic example of a PID loop is a thermostat. The thermostat is set to a certain temperature. The temperature of the process under control is measured. The error is then fed into a set of PID equations, and the resulting sum of those equations is then used to drive a heating or cooling element at a corresponding duty cycle.

Below is a diagram of the PID control:

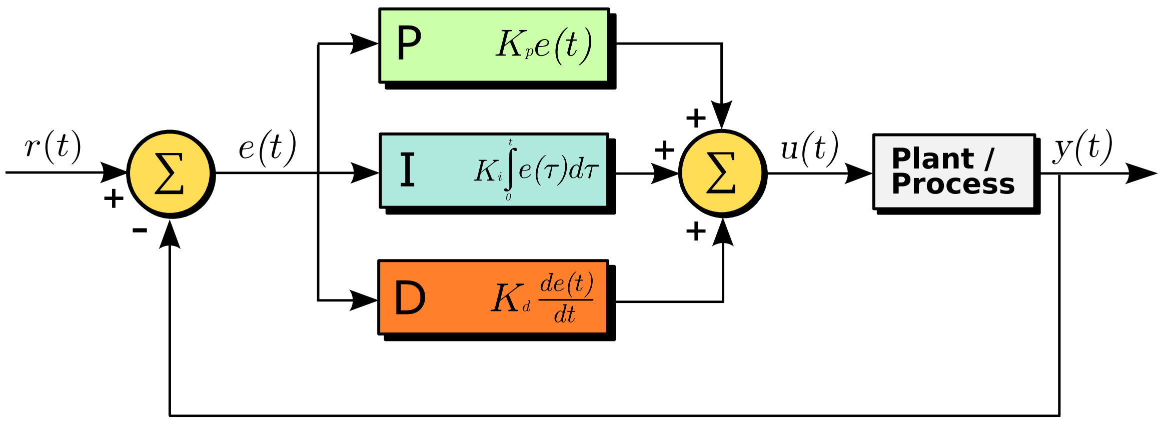

Figure 10: PID controller. Credit: Arturo Urquizo, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

Tuning the PID gains (Kp, Ki, Kd) can be a tricky matter and depends on the process being controlled. PID tuning is an extensive topic in itself. There is even an Arduino library for automatically tuning a PID, **PIDAutotuneLibrary**. Since we’re purely focused on the practical aspects of controlling a process, we will use generic or experimentally derived gains.

Build the following circuit.

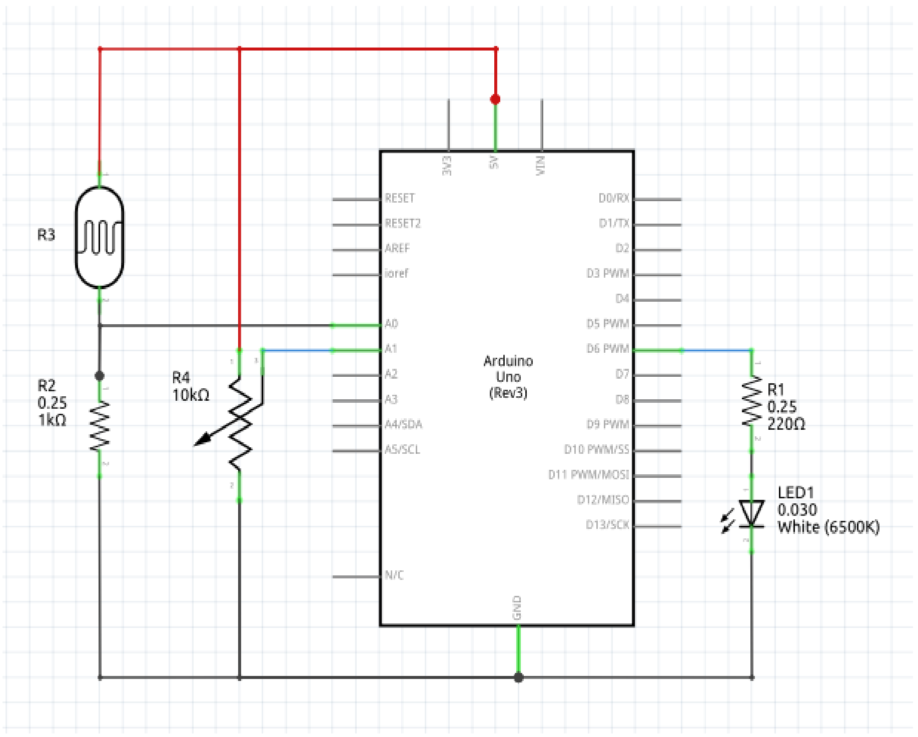

Figure 11: PID project schematic

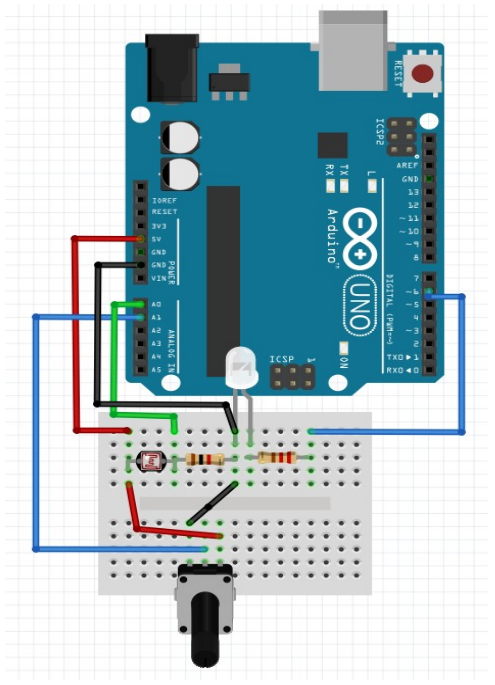

Figure 12: PID project circuit

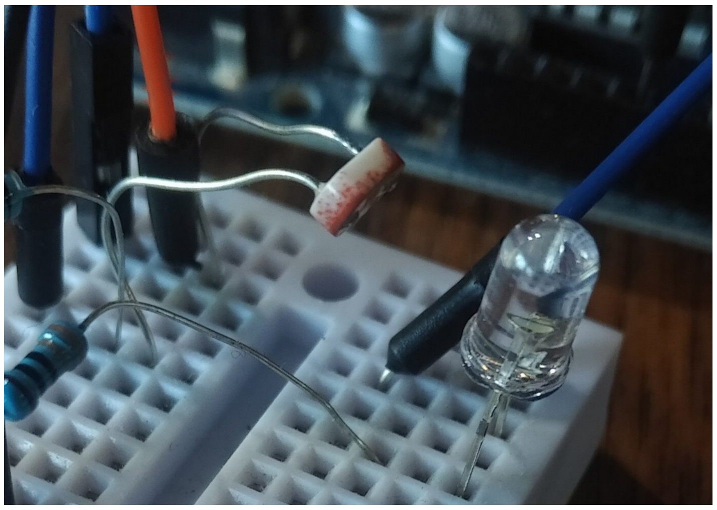

Figure 13: Orient the photocell to face the LED

In this project we will use the PID library by Brett Beauregard. Some classes and functions from the library that are relevant include:

PID(double* Input, double* Output, double* Setpoint,

double Kp, double Ki, double Kd, int ControllerDirection) – this constructs a PID object. It takes pointers to variables for the PID input, output, and setpoint. Passing a pointer to a variable to a function allows the function to change the variable’s value. The constructor takes the PID coefficients as well as a direction variable. For direction, DIRECT is for the direction of the PID control moves the output in relation to the input. REVERSE moves the output in the other direction.

Project specifications

We will build a very basic project to demonstrate the operation of a PID controller. The Arduino modulates the LED brightness that is read by the photocell. The setpoint is adjusted using a potentiometer. This is a slightly nonsensical project, but the necessary components are probably included with most Arduino kits. Make sure not to put the circuit in a brightly lit spot, or the ambient light will wash out the light from the LED.

Project code

Next type or copy/paste the following code into Arduino IDE:

```
// 1
#include <PID_v1.h>
#define PHOTO A0
#define POT A1
#define LED 6
#define UPDATE_INTERVAL 1000
#define SAMPLE_TIME 1

double setpoint, input, output;
double Kp = 0.01, Ki = 8, Kd = 0;
int previous = 0;
int now = 0;
// 2
PID pid_control(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

void setup() {
      Serial.begin(9600);
      pid_control.SetMode(AUTOMATIC);
      pid_control.SetSampleTime(SAMPLE_TIME);
      previous = millis();
}
// 3
void loop() {
      setpoint = analogRead(POT);
      input = analogRead(PHOTO);
      pid_control.Compute();
      analogWrite(LED, output);
      now = millis();
// 4
      if (now - previous > UPDATE_INTERVAL) {
            Serial.println("Setpoint: " + String(setpoint));
            Serial.println("Input: " + String(input));
            Serial.println("Output: " + String(output) + '\n');
            previous = millis();
      }
}
```

Code explanation

1. Here we set constants for pin assignments. The Arduino prints information to the serial port every UPDATE_INTERVAL milliseconds. SAMPLE_TIME determines how often the internal state of the PID updates, measured in seconds. We create variables for running the PID and constants for the PID coefficients. These coefficients were derived experimentally. The variables *now* and *previous* hold internal clock values so that the Arduino knows when to print to the serial port. *previous* is the last time there was a print to the serial port.

2. Here we create the PID object and pass pointers to the variables *setpoint*, *input*, and *output*, as well as the PID coefficients. DIRECT is for the direction of the PID control moves the output in relation to the input. REVERSE moves the output in the other direction. The PID is set to AUTOMATIC mode, as opposed to MANUAL, which starts the PID. The sample time is set. *previous* is assigned a value from millis() to get started.

3. Each loop, the setpoint and input are read from the analog ports. The PID's compute function needs to run ever loop. An updated output value is written to the analog out.

A few things should be noted about this example. Since it is highly simplified, there are several issues. One is that the ambient light is not taken into account. Second, we are dealing with raw analog readings without scale or proper units. Third, the output LED can only create a limited amount of light when directly driven from a digital port. If the setpoint, as read from the potentiometer, is set too high for the LED, the PID will not reach the setpoint.

now is updated with the latest timestamp from millis().

4. If enough time has elapsed since the last time the PID information was printed, the current information is printed.

Project review

The lab has given a functional introduction to creating a PID control with Arduino. For more information about using this PID library, check out the Github page at

https://github.com/br3ttb/Arduino-PID-Library. The library's author has written several blog articles about how the code was written.

Conclusion

In this course we have built four controls with Arduino that can be used in many situations. Arduino simplifies building controls, allowing you to quickly build a prototype or something for the lab.

Thank you for taking this course. I hope you got some value from it. Feel free to email any feedback you have.