State machine with Arduino

This paper is to introduce the concept of state machines and how to implement them in an Arduino Sketch.

The use of a state machine allows us to design easily more and more complex tasks. We will start with very simple machines and work our way into more challenging projects.

# What is a state machine?

It is a way to describe what something does, depending on what state it is presently. As an example, we could use a LED. A LED can only be in one of two states: ON or OFF. Each state is mutually exclusive of the other (a LED cannot be simultaneously ON and OFF.

Obviously, we would want to turn the LED ON or OFF at will. To do this, we use transitions. We will use the first Sketch that is used to present Arduino: Blink. Before we start writing the sketch, we will make a diagram of our state machine:

Wait 1,000 ms

Wait 1,000 ms

With the two circles, we see that the LED has two states: ON and OFF. The arrows show the transitions between each state. When in state ON, after 1000 ms, transition to state OFF. When in state OFF, after 1000 ms, transition to state ON.

Before starting to write a Sketch, we will always sit down with pen and paper and draw our diagram.

To remember in which state the machine currently is and other stuff, we will add attributes to the machine.

# Translating the diagram into an Arduino Sketch.

First, we will declare the LED’s attributes (before the setup() part of the sketch):

enum ledStates {ON, OFF}; //Give names to the LED’s states

//Our blink()’s attibutes

byte ledPin = 13; //We use the LED that is onboard on Arduino’s pin 13

byte ledStates ledCurrentState; //Make a global variable to hold the switches’ current state

int blinkTime = 1000; //Set how long between status changes

In the setup() section of the sketch, we will prepare the digital pin 13 for output. We will also decide what is the state that the LED is in when we start the sketch, by setting it’s ledCurrentState attribute. We will start with the LED on.

void setup() {

pinMode(ledPin, OUTPUT);

ledCurrentState = ON;

}

We need to create a new state machine, using a switch construct:

void blink() {

switch (ledCurrentState) { //Depending on the current state

case ON: { //ON

digitalWrite(ledPin, HIGH); //Make sure that the LED is ON

delay(blinkTime); //Wait for 1000ms

ledCurrentState = OFF; //Change the state to OFF

break; //Get out of the switch construct

}

case OFF: { //OFF

digitalWrite(ledPin, LOW); //Make sure that the LED is OFF

delay(blinkTime); //Wait for 1000ms

ledCurrentState = ON; //Change the state to ON

break; //Get out of the switch construct

}

}

}

The switch construct allows us to specify what is to be done when the machine is in that state and ignore the other states.

It contains:

* the instructions on what is to be done if in that state,
* the instructions to find out if we need to change state (or making it so) and changing the state if needed.

In the loop section, we just need to call blink():

void loop() {

blink();

}

Let’s say that we want to have the LED to stay on 1000 ms, and to stay off 500ms. We would change our diagram to look like this:

Wait 1,000 ms

Wait 500 ms

For our first state machine, we only had one attribute to tell how long to wait between state changes. It is now obvious that we need two of them. The easiest way to go about it is to change the single attribute with a table that will contain both the ON time and the OFF time. So we will remove:

blinkTime = 1000;

And replace it with:

blinkTimes[2] = {1000, 500};

Then our blink() method would only need to have two lines changed:

void blink() {

switch (ledCurrentState) {

case ON: {

digitalWrite(13, HIGH);

delay(blinkTimes[0]);

ledCurrentState = OFF;

break;

}

case OFF: {

digitalWrite(13, LOW);

delay(blinkTimes[1]);

ledCurrentState = ON;

break;

}

}

}

# Blink without delay()

Our blink() method has a serious flaw. It has the processor going idle for 99.99% of the time doing it’s delay() thing. That prevents the sketch to do other tasks in the meantime.

To free the processor to do other things, we will add another attribute to our LED machine. A chronometer. Arduino has an internal clock that counts the number of milliseconds elapsed since the Sketch started. We will use that clock to count the milliseconds between state changes instead of the delay. A call to millis() is the same as asking time.

Suppose that we want to have a state that’s duration is 1000ms. We could have an attribute called “chrono”, and ask it to be equal to Arduino’s clock (Same as setting chrono to “now”), and let it be our “Start counting signal”. We then let Arduino do other things until some point where the difference between Arduino’s clock and our “Start counting signal” goes equal or beyond 1000ms.

That, in Arduino’s terms would be:

First, establish the start of the time limit:

chrono = millis();

Then find out if we are at or beyond that limit:

if(millis() – chrono) >= 1000) {

//do something

}

Now, we will modify our blink() method to be super-efficient, requiring only millionths of a seconds to do its job instead of the 1½ seconds it used to.

Before the setup() we add this attribute to our switch state machine:

NOTE : Unlike standard longs unsigned longs won't store negative numbers, making their range from 0 to 4,294,967,295. Since millis() returns an unsigned long value, chrono will also have to be an unsigned long.

unsigned long chrono = millis();

The setup() part of our sketch will remain the same.

We will modify the blink() method as follows:

void blink() {

switch (ledCurrentState) {

case ON: {

digitalWrite(ledPin, HIGH);

if(millis()- chrono) >= blinkTimes[ON]) {

chrono = millis();

ledCurrentState = OFF;

}

break;

}

case OFF: {

digitalWrite(ledPin, LOW);

if(millis()- chrono) >= blinkTimes[OFF]) {

chrono = millis();

ledCurrentState = ON;

}

break;

}

}

}

## Making it more efficient

Looking at this code, we see that there is redundancy (code is almost identical):

digitalWrite(ledPin, HIGH);

if(millis()- chrono) >= blinkTimes[ON]) {

chrono = millis();

ledCurrentState = OFF;

}

digitalWrite(ledPin, LOW);

if(millis()- chrono) >= blinkTimes[OFF]) {

chrono = millis();

ledCurrentState = ON;

}

We will write a method that takes care of that:

void handleState(byte pinState, byte state, byte nextState) {

digitalWrite(ledPin, pinState);

if(millis()-chrono >= blinkTimes[state]) {

chrono = millis();

ledCurrentState = nextState;

}

}

This method is our blink() method’s helper. We will only call it with the things that differ from what the ON state has to do and what the OFF state has to do. What is different between both?

* Line 1: it is either HIGH or LOW (pinState)
* Line 2: the state we are actually in ON or OFF (state)
* Line 3: nothing
* Line 4: The next state is either OFF or ON (nextState)

The blink() then becomes:

void blink() {

switch (ledCurrentState) {

case ON: { handleState(HIGH, ON, OFF); break; }

case OFF: { handleState(LOW, OFF, ON); break; }

}

}

# Adding a second LED

Now, we will add a second LED that will blink at a different rate.

GREEN LED RED LED

Wait 1,000 ms

Wait 500 ms

Wait 300 ms

Wait 200 ms

Since we will have two states machines that are almost identical, it would make sense that our blink method would be changed to handle more than one LED. We will place all the attributes **in tables** like we did with blinkTime[]. All the attributes for the green LED will be in column 0, and the attributes for the red LED will be in the column 1.

Before setup() we have:

enum ledStates {ON, OFF};

#define GREEN 0

#define RED 1

byte ledPins[2] = {12, 13};

byte ledStates ledsStates[2];

unsigned long chronos[2];

int blinkTimes[2][2] = {1000, 500,

300, 200};

In the setup() we have

void (setup() {

pinMode(ledPins[GREEN], OUTPUT);

pinMode(ledPins[RED], OUTPUT);

ledsStates[GREEN] = ON;

ledsStates[RED] = ON;

chronos[GREEN] = millis();

chronos[RED] = millis();

}

We will add another parameter to our handleState() routine to specify which LED is currently being processed. The method that handles the states becomes:

void handleState(byte ledId, byte pinState, int state, byte nextState) {

digitalWrite(ledPins[ledId], pinState);

if(millis()-chronos[ledId] >= blinkTimes[ledId][state]) {

chronos[ledId] = millis();

ledsStates[ledId] = nextState;

}

}

In the loop() we have:

void loop() {

switch (ledsStates[GREEN]) {

case ON: { handleState(GREEN, HIGH, ON, OFF); break; }

case OFF: { handleState(GREEN, LOW, OFF, ON); break; }

}

switch (ledsStates[RED]) {

case ON: { handleState(RED, HIGH, ON, OFF); break; }

case OFF: { handleState(RED, LOW, OFF, ON); break; }

}

}

Which could also be done with a for( ; ; ) loop:

void loop() {

for (byte led = GREEN ; led <=RED ; led++) {

switch (ledsStates[led]) {

case ON: { handleState(led, HIGH, ON, OFF); break; }

case OFF: { handleState(led, LOW, OFF, ON); break; }

}

}

}

# Reading a switch

digitalRead(pin) ==HIGH

digitalRead(pin) ==LOW

Here, we have a normally open switch that is connected to the Arduino. A normally open switch conducts current only when it is pressed. Let’s examine the four states of that switch. **(Don’t forget the pulldown resistor)**

* At the start, it is in the state OPEN.
* When (digitalRead(pin) == HIGH), it means that it is no longer OPEN. We will call that state RISING
* Follows immediately the CLOSED state (it is not rising anymore)
* When (digitalRead(pin) == LOW), it means that it is no longer CLOSED. We will call that state FALLING
* Follows immediately the OPEN state again (it is not falling anymore)

Before the setup() we will write:

byte switchPin = 2;

enum switchStates {OPEN, RISING, CLOSED, FALLING};

switchStates switchCurrentState;

In the setup() we will write:

void setup() {

pinMode(switchPin, INPUT);

switchCurrentState = OPEN;

}

We can now create the readSwitch() state machine

void readSwitch() {

switch (switchCurrentState) {

case OPEN: { if(digitalRead(switchPin) == HIGH) switchCurrentState = RISING; break; }

case RISING: { switchCurrentState = CLOSED; break; }

case CLOSED: { if(digitalRead(switchPin) == LOW) switchCurrentState = FALLING; break;}

case FALLING: { switchCurrentState = OPEN; break; }

}

}

Very often, we just want to know if a switch has been pressed to trigger an event. The user presses the switch then releases it. That is what is often referred to as a click. For this to happen, the switch (originally in a OPEN state) will go to the RISING state when the user presses the switch, then go and stay in the CLOSED state until the user releases the switch, at which point it will be in the FALLING state. This is exactly when we want to take action.

In the loop(), we could write:

void loop() {

readSwitch();

if(switchCurrentState == FALLING) {

//Do something

}

}

NOTE: Of course, if the switch’s state is used to trigger other events, do not invoke “readSwitch()” again in the same loop, as it would bring it in the next state (OPEN). Just use the “if” construct.

# Using a switch with either the internal pullup resistor or an external pulldown resistor.

With a pullup resistor, the values returned when the switch is pressed are reverse than with a pulldown resistor:

Pressed Released

Pulldown: HIGH LOW

Pullup: LOW HIGH

Before the setup() we will add an attribute that allows us to specify the type of connection:

byte switchPin = 2;

enum switchStates {OPEN, RISING, CLOSED, FALLING};

switchStates switchCurrentState;

bool isPullup = true;

The first change that we have to do, is in the setup() part of the sketch:

We will replace:

pinMode(switchPin, INPUT);

with:

pinMode(switchPin, INPUT\_PULLUP);

Next, we can see that the only thing different, is the transition conditions are reversed.

digitalRead(pin) ==LOW

digitalRead(pin) ==HIGH

# Using the switch to toggle a LED

digitalRead(pin) ==HIGH

digitalRead(pin) ==LOW

**LED SWITCH**

True

True

We want to toggle the LED whenever we click the switch.

We need:

* a LED state machine,
* a Switch state machine,
* and a method that we will call toggle() that sends the message to the LED state machine that the Switch state machine is in the FALLING state.

Before the setup() part of the sketch, we need to create the attributes for the two state machines:

ledPin = 13;

enum ledStates {ON, OFF};

ledStates ledCurrentState;

switchPin = 2;

enum switchStates {OPEN, RISING, CLOSED, FALLING};

switchStates switchCurrentState;

The LED state machine will look like this:

void toggleLed(){

switch (ledCurrentState) {

case ON: { digitalWrite(ledPin, LOW); ledCurrentState = OFF; break; }

case OFF: { digitalWrite(ledPin, HIGH); ledCurrentState = ON; break; }

}

}

Our switch state machine remains the same:

void readSwitch() {

switch (switchCurrentState) {

case OPEN: { if(digitalRead(switchPin) == HIGH) switchCurrentState = RISING; break; }

case RISING: { switchCurrentState = CLOSED; break; }

case CLOSED: { if(digitalRead(switchPin) == LOW) switchCurrentState = FALLING; break;}

case FALLING: { switchCurrentState = OPEN; break; }

}

}

We then create a method that links the LED state machine and the Switch state machine together:

void toggle() {

readSwitch();

if(switchCurrentState == FALLING) toggleLed();

}

The loop() will contain this:

void loop() {

toggle();

}

# About debouncing

We should never use digitalRead() alone. This function only reads the pin once. Switches bounce as they get closed or open. It is always a good idea to debounce the switch. There are many libraries that will do that for you. Anyone will do. I happen to have written one that is called “edgeDebounceLite”.

You can download it here: <https://github.com/j-bellavance/EdgeDebounceLite>

In the beginning of the sketch we write:

#include <EdgeDebounceLite.h}

EdgeDebounceLite debounce;

Now, instead of using:

aVariable = digitalRead(switchPin);

We use:

aVariable = debounce.pin(switchPin)