State machine with Arduino (Part 3)

# The elevator state machine

TABLE OF CONTENTS

[What we need 2](#_Toc495320058)

[The cabin stateMachine 2](#_Toc495320059)

[The code 3](#_Toc495320060)

[Handling the requests 4](#_Toc495320061)

[Controling the direction of the cabin 7](#_Toc495320062)

[Controlling the cabin motor 8](#_Toc495320063)

[Controling the cabin door 9](#_Toc495320064)

## What we need

Let’s say that we want to operate an elevator for an 8 level building. We will need:

* an elevator cabin with its state machine;
* a cabin control panel with 8 story selector switches (drop off requests);
* up and down switches for each floor (pickup requests) (except the first floor *up only* and the last floor *down only*) (total 14 switches);
* one proximity switch to asses that the door is closed;
* eight proximity switches wired in parallel at the floor level to stop the cabin at the proper level;
* seven proximity switches wired in parallel at midpoint between levels to start decelerating the cabin;
* a 360 degrees servo cabin motor that can reverse its rotation.
* a 180 degrees servo to open and close the doors.

## The cabin stateMachine

Requests for same floor

TRUE

Requests for

other floors

Near next floor

ELSE

Requests

for this floor

TRUE

More requests for

same direction

ELSE

Requests for

other direction

ELSE

Stopped

Door closed confirmed

TRUE

TRUE

UpToSpeed

Start

## The code

enum ElevatorStates { REMAIN\_ON\_THIS\_FLOOR, SET\_DIRECTION, ACCELERATE,

MOVE\_DIRECTION, STOP\_DECISION, PASS\_NEXT\_FLOOR,

STOP\_ON\_THIS\_FLOOR, OPEN\_DOOR, WAIT\_FIVE\_SECONDS,

CLOSE\_DOOR, DOOR\_CLOSED, NEXT\_PHASE\_DECISION,

CHANGE\_DIRECTION};

ElevatorStates elevatorState = REMAIN\_ON\_THIS\_FLOOR;

bool requestsFound(int direction) {}

bool requestsForOtherFloors() {}

bool requestsForSameFloor() {}

bool requestsForThisFloor() {}

void setDirection() {}

void accelerate() {}

bool upToSpeed() {}

void moveElevator() {}

bool nearNextFloor() {}

void decelerate() {}

bool stopped() {}

void openDoor() {}

void closeDoor() {}

bool doorClosedConfirmed() {}

void changeDirection() {}

void elevator() {

switch (elevatorState) {

case REMAIN\_ON\_THIS\_FLOOR: {

cabinDirection = STILL;

if (requestsForOtherFloors()) elevatorState = SET\_DIRECTION;

if (requestsForSameFloor()) elevatorState = OPEN\_DOOR;

}

case SET\_DIRECTION: {

setDirection();

elevatorState = ACCELERATE;

}

case ACCELERATE: {

accelerate();

if (upToSpeed()) elevatorState = MOVE\_DIRECTION;

}

case MOVE\_DIRECTION: {

moveElevator();

if (nearNextFloor()) elevatorState = STOP\_DECISION;

}

case STOP\_DECISION: {

if (requestsForThisFloor()) elevatorState = STOP\_ON\_THIS\_FLOOR;

else elevatorState = MOVE\_DIRECTION;

}

case STOP\_ON\_THIS\_FLOOR: {

decelerate();

if (stopped()) elevatorState = OPEN\_DOOR;

}

case OPEN\_DOOR: {

openDoor();

eraseRequestsForFloor(cabinPosition);

elevatorState = WAIT\_FIVE\_SECONDS;

openDoorChrono = millis();

}

case WAIT\_FIVE\_SECONDS: {

if (millis() - openDoorChrono > 5000) elevatorState = CLOSE\_DOOR;

}

case CLOSE\_DOOR: {

closeDoor();

if (doorClosedConfirmed()) elevatorState = DOOR\_CLOSED;

}

case DOOR\_CLOSED: {

if (requestsFound(cabinDirection)) elevatorState = MOVE\_DIRECTION;

else elevatorState = NEXT\_PHASE\_DECISION;

}

case NEXT\_PHASE\_DECISION: {

if (requestsFound(-cabinDirection)) elevatorState = CHANGE\_DIRECTION;

else elevatorState = REMAIN\_ON\_THIS\_FLOOR;

}

case CHANGE\_DIRECTION: {

changeDirection();

elevatorState = MOVE\_DIRECTION;

}

}

}

## Handling the requests

Let’s take a look at how many switches we will need to set the requests:

* Cabin panel drop off switches 8
* Up / Down floor pickup switches 14

**Total : 22**

That is a lot of switches. Instead of using digital pins, we will have voltage divider circuits that use analog pins.

The AnalogSwitches Library[[1]](#footnote-1) will help us to manage all those switches.

#include <AnalogSwitches.h>

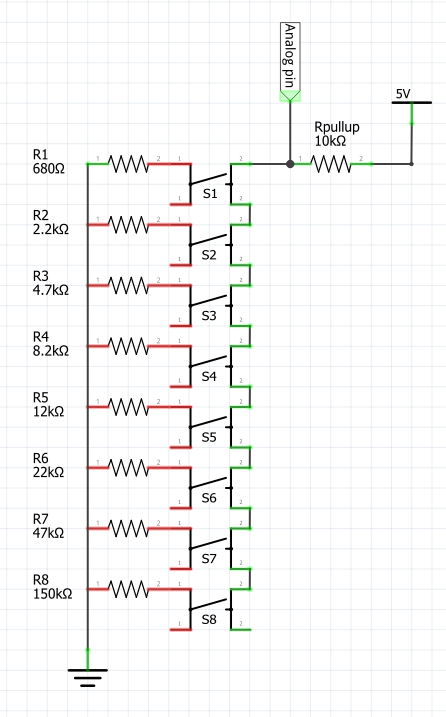
AnalogSwitches switchBank(3);

Our switch banks will contain 3 banks this way:

* Pin A0: Cabin panel drop off switches; .addSwitches(1, A0, 8, 100);
* Pin A1: Going up floor pickup switches; .addSwitches(2, A1, 8, 200);
* Pin A2: Going down floor pickup switches; .addSwitches(3, A2, 8, 300);

The .getIndex() method will return 101..108, 202..208 or 301..307

Typical schematic for the 3 voltage divider circuits.



Circuit for pin A0 has all 8 switches & 8 resistors.

Values returned will be 101, 102, 103, 104, 105, 106, 107, 108

Circuit for pin A1 has switch1 & resistor1 missing.

Values returned will be 202, 203, 204, 205, 206, 207, 208

Circuit for pin A2 has switch8 & resistor8 missing.

Values returned will be 301, 302, 303, 304, 305, 306, 307

The switches will trigger requests for stops at certain floors. We can have three kinds of requests:

* Drop off requests, generated by the 8 switches in the cabin’s control panel.
* Pick up requests for people that want to go down, generated by the DOWN switches at each floor (except for the first floor that does not have that switch.
* Pick up requests for people that want to go up, generated by the UP switches at each floor (except for the last floor that does not have that switch.

We will use three boolean tables to store those requests:

bool dropOffRequests[8]; //ID [101..108], int(ID/100) is 1, (ID%100)-1 is [0..7]

bool pickUpGoingDownRequests[8]; //ID [202..208], int(ID/100) is 2, (ID%100)-1 is [1..7]

bool pickUpGoingUpRequests[8]; //ID [301..307], int(ID/100) is 3, (ID%100)-1 is [0..6]

The handleSwitches function will fill the three tables:

void handleSwitches() {

int switchIndex = switchBank.getIndex();

if (switchIndex == 0) return;

switch (int(switchIndex / 100)) {

case 1: { dropOffRequests[(switchIndex % 100) -1] = true; break; }

case 2: { pickUpGoingUpRequests[(switchIndex % 100) -1] = true; break; }

case 3: { pickUpGoingDownRequests[(switchIndex % 100) -1] = true; break; }

}

}

In our Elevator state machine, we have four decision making transitions concerning those requests and some of them have to be made according to the direction we are moving (up or down). So we need:

enum CabinDirections {DOWN = -1, STILL = 0, UP = 1};

CabinDirections cabinDirection = STILL;

int cabinPosition = 0; //We start on the ground floor

We will also need a flag that tells us if the request is on the same floor another one if the request is for another floor than the cabin’s position and a variable to hold the floor number for that request:

bool requestIsOnTheSameFloor;

bool requestIsOnAnotherFloor;

byte requestIsForFloor;

We will look for requests starting from the cabin position to the top floor if we are going up, or to the ground floor if we are going down. We will invert the search if we want to check for requests in the other direction.

bool requestsFound(int direction) {

requestIsOnTheSameFloor = false;

for (int i = cabinPosition ; i < 8 && i >= 0 ; i += direction) {

if (dropOffRequests[i] == true) {

requestIsForFloor = i;

if(i == cabinPosition) requestIsOnTheSameFloor = true;

else requestIsOnAnotherFloor = true;

return true;

}

if (pickUpGoingUpRequests[i] == true) {

requestIsForFloor = i;

if(i == cabinPosition) requestIsOnTheSameFloor = true;

else requestIsOnAnotherFloor = true;

return true;

}

if (pickUpGoingDownRequests[i] == true) {

requestIsForFloor = i;

if(i == cabinPosition) requestIsOnTheSameFloor = true;

else requestIsOnAnotherFloor = true;

return true;

}

}

return false;

}

bool requestsForOtherFloors() {

if (requestsFound(UP) == true && requestIsOnAnotherFloor) return true;

if (requestsFound(DOWN) == true && requestIsOnAnotherFloor) return true;

return false;

}

bool requestsForSameFloor() {

if (requestsFound(UP) == true && requestIsOnTheSameFloor) return true;

if (requestsFound(DOWN) == true && requestIsOnTheSameFloor) return true;

return false;

}

The next function is different from the others, since we are not interested by requests that require a pickup that is not in for our current direction, and we check only for the current cabin position.

bool requestsForThisFloor() {

if (dropOffRequests[cabinPosition] == true) return true;

switch (cabinDirection) {

case DOWN: if (pickUpGoingDownRequests[cabinPosition] == true) return true;

case UP: if (pickUpGoingUpRequests[cabinPosition] == true) return true;

}

return false;

}

Finally, when we open the door, we want to have all the requests for that floor to be erased:

void eraseRequestsForFloor(byte floor) {

dropOffRequests[floor] = false;

pickUpGoingUpRequests[floor] = false;

pickUpGoingDownRequests[floor] = false;

}

## Controling the direction of the cabin

We have already seen those three lines in the *handling the request* section:

enum CabinDirections {DOWN = -1, STILL = 0, UP = 1};

CabinDirections cabinDirection = STILL;

int cabinPosition = 0; //We start on the ground floor

At first, when the cabin is told to “remain on the same floor”, the direction is STILL. When a request is made, we need to know which way to go to answer that request.

void setDirection() {

if (requestIsForFloor > cabinPosition) cabinDirection = UP;

else cabinDirection = DOWN;

}

Then, the cabin will keep going in that direction until all the requests have been answered. At which point it will either “remain on the same floor” or “change direction”.

void changeDirection() {

if (cabinDirection == DOWN) cabinDirection = UP;

else cabinDirection = DOWN;

}

## Controlling the cabin motor

We will use a 360 degrees servo motor, connected to a PWM digital pin. We will use <servo.h> as the cabin motor, on pin 3. For this sketch, we will use 90 degrees as our stand still position. We will set the maximum speed at 85 and the minimum speed at 2. Changing the direction is only a matter of using the cabinDirection variable that we saw earlier.

#include <Servo.h>

Servo cabinMotor;

byte cabinMotorPin = 3; //PWM capable

byte currentMotorSpeed = 0;

byte motorMaxSpeed = 85;

byte motorSlowSpeed = 2;

byte motorStandStill = 90; //We adjust this value if needed

We will accelerate up to full speed in the direction we want to go, and keep going until the cabin state machine decides that we have to stop at a certain level. That decision is made midway between two floors with the help of seven proximity switches that are connected in parallel. If we need to stop, we decelerate to the minimum speed, and keep going at that speed until one of the eight proximity switches placed at floor level triggers. At that point, we stop the motor.

The proximity switches will be wired in parallel to pins 4 (midpoint) and 5 (floor level);

byte atMidpointPin = 4; //Switches in parallel

byte atFloorPin = 5; //Switches in parallel

We will use EdgeDebounceLite[[2]](#footnote-2) to debounce the pins.

#include <EdgeDebounceLite.h>

EdgeDebounceLite debounce;

We read the switch with debounce.pin(pin)

The function for this are:

void accelerate() {

if (currentMotorSpeed < motorMaxSpeed) {

currentMotorSpeed++;

cabinMotor.write(motorStandStill + (currentMotorSpeed \* cabinDirection));

}

}

bool upToSpeed() {

if (currentMotorSpeed == motorMaxSpeed) return true;

else return false;

}

void moveElevator() {

cabinMotor.write(motorStandStill + (currentMotorSpeed \* cabinDirection));

}

bool nearNextFloor() {

if (debounce.pin(atMidpointPin) == LOW) {

cabinPosition += cabinDirection;

return true;

}

}

void decelerate() { //and stop on this floor

if (currentMotorSpeed > motorSlowSpeed) {

currentMotorSpeed--;

cabinMotor.write(motorStandStill + (currentMotorSpeed \* cabinDirection));

}

if (debounce.pin(atFloorPin) == LOW) {

currentMotorSpeed = motorStandStill;

cabinMotor.write(motorStandStill);

}

}

bool stopped() {

if (currentMotorSpeed = motorStandStill) return true;

}

## Controling the cabin door

The cabin door will be opened and closed by a 180 degrees servo motor connected to pin 6. When the motor is at 130 degrees, the door is open, and when it is at 50 degrees, the door is closed. For security reasons, we will have a proximity switch on the door to asses that the door is really closed before going on. This switch will be on pin 7.

Servo doorMotor;

byte doorMotorPin = 6; //PWM capable

byte doorOpenPosition = 130;

byte doorClosedPosition = 50;

byte doorClosedPin = 7;

The functions to control the door are:

void openDoor() {

doorMotor.write(doorOpenPosition);

}

void closeDoor() {

doorMotor.write(doorClosedPosition);

}

bool doorClosedConfirmed() {

if (debounce.pin(doorClosedPin) == LOW) return true;

}

1. Find it at : <https://github.com/j-bellavance/AnalogSwitches> [↑](#footnote-ref-1)
2. You can find it here : <https://github.com/j-bellavance/EdgeDebounceLite> [↑](#footnote-ref-2)