# State machine with Arduino (Part 3)

## The elevator state machine

TABLE OF CONTENTS

## Overview

We want to operate an elevator for an 8 level building (ground floor plus 7 stories).

Since we don't want to build a real elevator, we will use the serial monitor to see what our elevator does, and to push the buttons.

We will take requests from users and drive the elevator cabin to satisfy those requests. Requests are made on each floor with two buttons: One for going up and another one for going down. Requests are also made with buttons inside the cabin to say on what floor we want to be dropped off. There is one button per floor.

To simulate the position of the cabin according to floor level, we will use a scale where 1000 units represent one floor. So the elevation of the cabin will be between 0 and 7000. Going up, from stop, at floor level (x000), the cabin will accelerate for a while until elevation is x045, then run at full speed until elevation is x955, and then decelerate until full stop at the next floor level (y000). Going down, changes will occur at (y000 ➢ x955 ➢ x045 ➢ x000).These elevations will be printed on the serial monitor.

The cabin is standing still until it receives a request. If it is for the same floor, the cabin only has to open its doors. If it is for another floor, it will move in that direction. Once it starts moving in a direction, it will answer all requests from people going in that direction. When it gets to the last request in that direction, it will answer requests for the other direction. It will do so until there are no more requests and will remain on that floor, with its doors closed.

When we reach the floor, we will open the doors, we will wait there for five seconds; and then close the doors.

We will also simulate a sensor to make sure that the door is closed before going up or down. That should make it easier to add buttons on the cabin panel to open and to close the doors, and/or to place sensors on the doors so that people going in have a chance to keep it open. It will not be part of the project, but if you feel like doing it, this feature should make it easier.

# Handling requests

## Remembering the requests

In the cabin, we have a control panel with 8 buttons, one for each floor. They will be referred to as “Drop-off” requests.

On each floor, we have two buttons, one for going up, and another one for going down. They will be referred to as pickup requests.

To remember where to stop, we will put them in a table. It will be a two dimension table. There will be 8 columns, one for each floor, and 3 lines, one for each type of requests.

```
bool requests[8][3] = {false, false, false, false, false, false, false, false,  //drop-off
                       false, false, false, false, false, false, false, false,  //pickup going up
                       false, false, false, false, false, false, false, false}; //pickup going down
```

## Typing commands in the Serial monitor to simulate the buttons

To simulate pressing the buttons, we will use the serial monitor. The commands will be made of 2 characters:

- The first one will be a letter: **C** for Cabin, **U** for going Up, and **D** for going Down.
- The second one will be a number between 0 (ground level) and 7 (top level).
- The third one will be the enter key. The serial monitor has to be set to the “Carriage Return” mode
- The ground level has only one button for going up, so D0 will be rejected.
- The top level has only one button for going down, so U7 will also be rejected.
- Any other combination will be rejected.

## The code to read the Serial monitor

```
bool readMonitor() {
//Our variables
  static char buf[3];   //A buffer for our commands: 2 characters plus the carriage return
  static byte i = 0;    //In what position the next character will be placed in the buffer
  byte flr;             //Table column index: between 0 and 7
  byte requestType;     //Table row index: 0, 1 or 2

//Here, we read the Serial Monitor
  if (Serial.available()>0){
     buf[i] = Serial.read();
     i++;
  }

//Here, we check if the last character that we read buf[2] was a carriage return
  if (int(buf[2]) == 13) {

  //We reset the buffer pointer to be ready for the next command and erase the carriage return
    i = 0; buf[2] = 0;

  //Here, we check the first character buf[0] and adjust the type of request
    switch (buf[0]) {
      case 'C': { requestType = 0; break; }
      case 'U': { requestType = 1; break; }
      case 'D': { requestType = 2; break; }
      default:  { return; }  //Reject if it is not 'C', 'U' or 'D'
    }
```

```
  //Here, we convert the second character buf[1] from ASCII ['0'..'7'] to an integer [0..8].
    flr = buf[1] - 48;  // '0' is ASCII 48

  //here, we check for the validity of the ccommand
    if (flr < 0 || flr > 7) return; //Has to be between ground floor and seventh floor
    if (flr == 7 && requestType == 1) return; //Can't go above seventh floor
    if (flr == 0 && requestType == 2) return; //Can't go below ground floor

//Since the command is valid, we set the request for [requested floor] and [request type] as true
    requests[flr][requestType] = true;
  }
}
```

## Questioning the request table

To know where to head next, the cabin has to be able to read the requests. The elevator will ask the following questions:
When it stands still:

- Are there any requests for same floor?
  (If so, the cabin will open the door)
- Are there any requests for other floors and if so, set the direction for me please?
  (If so, the cabin will accelerate in this direction)

When it is moving:

- Are there requests for drop-off or pickup going in my direction for the floor that I am about to reach?
  (If so, the cabin will stop on this floor and erase any requests made for that floor)
- Are there any drop-off requests for floors in my direction?
  (If so, the cabin will keep going in the same direction)
- I have no more drop-off requests. Are there any pickup requests for the remaining floors in my direction?
  (If so, the cabin will keep going in the same direction)

## The code

To answer those questions, we will need to know the direction of the cabin and its elevation:

```
int direction = 0; //Going down = -1, Standing still = 0 and going up = 1
int elevation = 0; //We start on the ground floor
int speed = 0;     //We will need this variable later on
```

For the first question, we will search for *any request for our floor*.

```
bool requestsForThisFloor() {
  flr = int(elevation / 1000);
  for (int i = 0 ; i < 3 ; i++) if (requests[flr][i]) return true;
  return false;
}
```

For the second question, we will search for *any request for any floor*. We will change the direction to get to that floor if we find a request. We will return true as soon as one is found. We will create a function that looks in a direction to see if there are requests between the next floor and the last floor. We use elevation + 45, because it can also be called by the deceleration decision transition. (Read further to the next question.)

```
bool requestsFor(byte direction) {
  for (byte flr = int((elevation + 45) / 1000) + direction ; flr >= 0 && flr <= 7 ; flr += direction)
    for (byte i = 0 ; i < 3 ; i++) if (requests[flr][i]) return true;
}
```

Then we will call this function

```
bool requestsForOtherFloors() {
  if (requestsFor( 1)) { direction =  1; return true; }
  if (requestsFor(-1)) { direction = -1; return true; }
  return false;
}
```

For the third question, we will search for *drop-off requests* and *pickup requests that are in our direction*. We check only for the current floor. We will not stop for requests going in the other direction unless there are no more requests going in that direction.

The actual call for this function will be when it is time to decelerate. When going up, the elevation will be – x955 – (x being the floor number) adding 45 to the elevation will give – y000 – (y being the next floor number) Dividing them by 1000 will yield the floor number. When going down, the calls are made at – x045 – adding 45 to this gives x090. Dividing this by 1000 and taking only the integer part will yield the floor number.

```
bool requestsForThisFloor() {
  bool requestSameDirection = false;
  flr = int((elevation + 45) / 1000); //Calls are made at deceleration elevations.
  if (requests[flr][0] == true) return true;                                    //drop-off
  if (!requestsFor(direction)) return true;
  switch (direction) {
    case  1: if (requests[flr][1] == true) {requestSameDirection = true; break;}//pickup going up
    case -1: if (requests[flr][2] == true) {requestSameDirection = true; break;}//pickup going down
  }
  if (requestSameDirection) return true;
  else                      return false;
}
```

For the fourth question, we will search for a *drop-off request* in our direction from the next floor to the last one (ground floor if going down or 7th floor if going up). We will return true as soon as one is found.

```
bool dropOffInMyDirection(int direction) {
  current = int(elevation / 1000) //Calls are made at floor level
  for (byte flr = current + direction; flr >= 0 && flr <= 7 ; flr += direction) {
    if (requests[flr][0]) return true;
  }
  return false;
}
```

For the fifth question, we will search for any *pickup request* in our direction from the next floor to the last one (ground floor if going down or 7th floor going up). We will return true as soon as one is found.

```
bool pickupsInMyDirection(int direction) {
  current = int(elevation / 1000) //Calls are made at flr level
  for (byte flr = current + direction; flr >= 0 && flr <= 7 ; flr += direction) {
    if (requests[flr][1]) return true;
    if (requests[flr][2]) return true;
  }
  return false;
}
```

We also need to erase all the requests for the current floor when the door is opened.

```
void eraseRequestsForThisFlr() {
  flr = int(elevation / 1000);
  for (int i = 0 ; i < 3 ; i++) (requests[flr][i]) = false;
}
```

## Moving the cabin

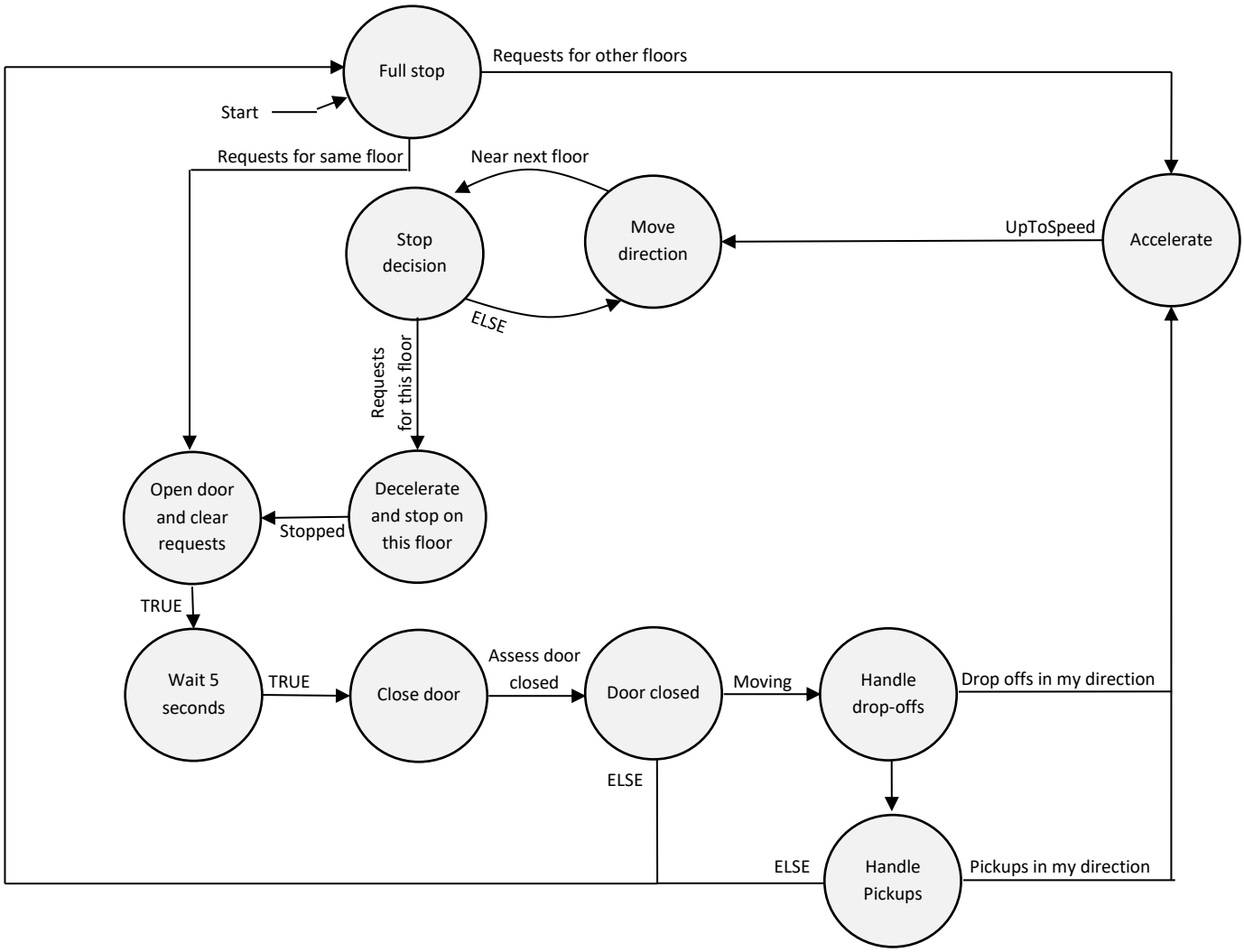
From **full stop**, if there are any requests for the same floor, it **opens the door**, **waits for 5 seconds** for people to get on. It will then **closees the doors** and assess that the **door is closed**. Since it is stopped, there is no direction to go to yet, and it will go back to **full stop**, waiting for the drop-off requests from those who entered the cabin.

From **full stop**, when the cabin has requests for other floors, it will **accelerate** until it is up to speed, **moving in that direction** at full speed. When the cabin is near the next floor, it has to **decide if it needs to stop**, and ask if there are requests for this floor. If it does, it will **decelerate** to stop on this floor. If it doesn't, it will keep **moving in the same direction**.

When the cabin stops, it **opens the doors and clears all the requests for this floor**, **waits for 5 seconds** for people to get off and for people to get on. It will then **close the doors** and assess that the **door is closed**. Since it was moving, it has to check for requests in the same direction.

It will first ask if there are drop-off requests in the same direction that it was going before stopping. If it does, it will **accelerate** in that direction. If not, it will ask if there are pick-up requests from floors in the same direction it was going. If so, it will also **accelerate** in that direction. If not, it will go in **full stop** mode, staying on this floor.

## The elevator State Machine



## The code

We already covered the five request handling transition functions, and the function to erase the requests. We will take a look at the other functions that the elevator will need.

## Handling transitions

First, we will take care of the functions that answer questions.

The cabin's elevation has 1000 units per floor. When accelerating, it will go from speed 0 to speed 10. Doing so, it will have moved by 45 units. The cabin will then move at speed 10 until it reaches the decision elevation. If it needs to stop, it will decelerate to speed 0. We will also need to confirm that the door is closed.

```
bool upToSpeed() {
  if (speed == 10) return true;
  else             return false;
}
bool nearNextFlr() {
  if (direction ==  1 && elevation % 1000 == 955
  ||  direction == -1 && elevation % 1000 ==  45) return true;
  return false;
}
bool stopped() {
  if (speed == 0) return true;
  else            return false;
}
bool moving() {
  if(direction != 0) return true;
  else               return false;
}
bool assessDoorClosed () {
  Serial.println("Door closed");
  return true;
}
```

## State actions

The next functions implement actions that have to be undertaken when the state machine is in a specific state. They could have been coded directly in the state machine, but placing them in functions makes it easier to maintain and to upgrade.

```
void accelerate {
  speed++;
  elevation += speed * direction;
  Serial.println(elevation);
}
void moveElevator() {
  elevation += speed * direction;
  Serial.println(elevation);
}
void decelerate() {
  speed--;
  elevation += speed * direction;
  Serial.println(elevation);
}
void openDoor() {
  Serial.println("Door open)";
}
void closeDoor() {
  Serial.println("Closing door)";
}
```

## The elevator State Machine code

```
enum ElevatorStates { FULL_STOP, ACCELERATE, MOVE_DIRECTION, STOP_DECISION,
                      DECELERATE, OPEN_DOOR, WAIT_FIVE_SECONDS, CLOSE_DOOR,
                      DOOR_CLOSED, HANDLE_DROPOFFS, HANDLE_PICKUPS};
ElevatorStates elevatorState = FULL_STOP;
unsigned long doorOpenChrono;

void elevator() {
  switch (elevatorState) {
    case FULL_STOP: {
      direction = 0;
      if      (requestsForSameFloor())  elevatorState = OPEN_DOOR;
      else if (requestsForOtherFloors()) elevatorState = ACCELERATE;
      break;
    }
    case ACCELERATE: {
      accelerate();
      if (upToSpeed()) elevatorState = MOVE_DIRECTION;
      break;
    }
    case MOVE_DIRECTION: {
      moveElevator();
      if (nearNextFloor()) elevatorState = STOP_DECISION;
      break;
    }
    case STOP_DECISION: {
      if (requestsForThisFloor()) elevatorState = DECELERATE;
      else                        elevatorState = MOVE_DIRECTION;
      break;
    }
    case DECELERATE: {
      decelerate();
      if (stopped()) elevatorState = OPEN_DOOR;
      break;
    }
    case OPEN_DOOR: {
      openDoor();
      eraseRequestsForThisFloor();
      elevatorState = WAIT_FIVE_SECONDS;
      doorOpenChrono = millis();
      break;
    }
    case WAIT_FIVE_SECONDS: {
      if (millis() - doorOpenChrono > 5000) elevatorState = CLOSE_DOOR;
      break;
    }
    case CLOSE_DOOR: {
      closeDoor();
      if (assessDoorClosed()) elevatorState = DOOR_CLOSED;
      break;
    }
    case DOOR_CLOSED: {
      if (moving())elevatorState = HANDLE_DROPOFFS;
      else         elevatorState = FULL_STOP;
      break;
    }
```

```
case HANDLE_DROPOFFS: {
      if (dropOffInMyDirection()) elevatorState = ACCELERATE;
      else                        elevatorState = HANDLE_PICKUPS;
      break;
    }
    case HANDLE_PICKUPS: {
      if (pickupsInMyDirection()) elevatorState = ACCELERATE;
      else                        elevatorState = FULL_STOP;
      break;
    }
  }
}
```

## setup & loop

```
void setup() {
  Serial.begin(9600);
  Serial.println(F("Full Stop"));
}

void loop() {
  readMonitor();
  elevator();
}
```