Intelligent public lighting for smart cities

João Coutinho - 89470 and João Porto - 89472

Instituto Superior Técnico, Av. Rovisco Pais 1, 1049-001 Lisboa
{j.afonso.coutinho,joaofporto}@tecnico.ulisboa.pt

Abstract. This works details an implementation of a modular and scalable system to control public lights, saving energy but preserving adequate levels of comfort and security. It simulates street lamps using LED cells and integrates multiple independent controllers with the use of a shared I2C bus and a predefined communication protocol. The controllers implement an independent clock synchronization system to track movement and achieve dynamic and reactive lighting.

1 Circuit Design

Figure 1 depicts a diagram of the developed circuit. In this particular example the *Red LED* has coordinates $(1, 0)$ and the *Green LED* has coordinates $(2, 0)$.

The buttons are connected to pins configured as $INPUT\_PULLUP$, which simplifies the circuit design by eliminating the need for capacitors or other additional components.

2 Software Architecture

2.1 Code structure

The program architecture for the Arduino controller is based on a Round Robin approach with interrupts. As the Arduino controller is responsible for 2 “LED lamps”, each lamp is represented by a *struct* that contains all its information. Each *struct* contains information about the pins for the lamp’s movement sensor, LED cell and circuit fault detection sensor, as well as variables depicting its target brightness and state. This allows for the same functions to run for both cells independently, allowing to simulate independent controllers with a single Arduino. This approach allows the addition or removal of cell lamps from the controller with minimal modifications to the code.

The setup function configures the digital pins and synchronizes the controller clock with all available neighbours. This synchronization is then demonstrated by blinking the built-in led of each Arduino every uneven second throughout the operation of the controller, allowing a visual representation of their clock synchronization.

The main loop checks the sensors of each cell independently, performs periodical checks for broken lamps and adjusts the lamp brightness accordingly. This loop is interrupted by I2C messages that change the cell’s state variables. The main loop will then act on those changes by reacting to the cells’ new state.

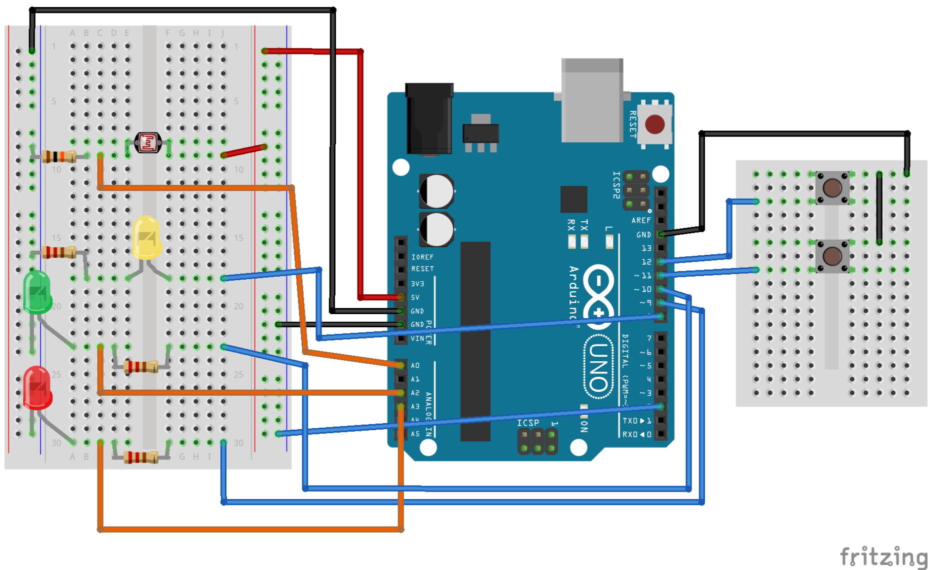

Fig. 1. Circuit diagram

2.2 Communication

Each message is also encoded in a *struct* containing the source and target cell coordinates, message code and a timestamp. Sending and parsing these messages from the I2C bus is handled by independent functions like $sendMessage$ and $receiveMessage$. This set of functions work like a communication API, allowing the controller to focus on high level logic and handle communication effortlessly. This also facilitates changing the communication bus to other medium or protocol maintaining the same code structure and logic.

Given that each Arduino controller is responsible for 2 lamp cells, it is often required to send messages from one cell to another handled by the same controller. The API abstracts this behaviour to the point where a cell only needs to send the *message struct* through the $sendMessage$ function and the API will either send it through the I2C bus or forward it directly to the neighbouring cell, according to whether that cell is controlled by the same Arduino.

3 Safety and Fault-Tolerance

3.1 Fault Detection

In order to detect faults in the system, there is a wire connecting each LED to an analog pin on the Arduino. The controller performs a periodic check to detect an open circuit - meaning failure of the LED - and when a fault is detected, the Arduino sends a message to the device at address $(0, 0)$, notifying it of the faulty LED. If the fault is corrected, an *OK* message is sent to the device at $(0,0)$ so it can keep track of the number of faulty LEDs exist in the system. Whenever there are faults present in the array of controllers, the device at $(0,0)$ signals a human operator with a *Yellow LED*. If a *FAULT* or *OK* message is received and the device receiving is not at $(0,0)$, the message is discarded. This helps prevent undefined behaviour when fault signals are sent to devices with coordinates different than $(0,0)$.

3.2 Network Delay

According to Equation 1, deduced in the Lab 2 work, a message of 9 bytes (the size of the messages in the defined protocol) will take about $0.801ms$ to reach its destination, as seen in Equation 2. Since the time unit for this project is $10ms$, the network delay is considered negligible.

$$Time(\mu s) = 95.33 \times Data(B) + 133.7 \tag{1}$$

$$9 \times 95.33 + 133.7 = 801.01\mu s \ = 0.801ms \tag{2}$$

4 Intelligent Functions

4.1 Brightness Reduction

When a light cell is set to a certain brightness level, this can only be reduced after that event has ended. This prevents such behavior like:

- (1, 0) $->$ detects movement and sets brightness to $FULL$.
- (2, 0) $->$ detects movements and warns (1,0) to set brightness to $HALF$.
- (1, 0) $->$ sets brightness to $HALF$ before the initial $FULL$ target brightness timeout ends.

This behavior can also occur if a *MOVEMENT* message is sent immediately after a *PREDICT* message. Since not all devices may implement this mechanism, our program always sends the *PREDICT* message after the *MOVEMENT* message in order to prevent this behavior.

4.2 Movement prediction

Each cell keeps track of movement in neighbouring cells independently with the help of an auxiliary array. When movement is detected within a cell, it checks if there was any recent movement in the neighbouring cells. In case there was, it calculates if the speed of that movement is above $5.6m/s$ and sends a message to the cell where that movement is predicted to go, e.g. if there was recent movement in a cell to the right, it sends a $PREDICT$ message to the cell to the left so it can turn on ahead of time.

5 Controller program

```
#include <Wire.h>

#define address(x, y) (x / 2) << 4 | y
#define time() (millis() - avgDelta)

// Schmitt variables
#define VREF 1000
#define H 10

// Duration of light: 10s
#define DURATION 3000

// Define maximum delta to consider movement (in ms)
#define CARD_DELTA 714  // delta between cardinal neighbours
#define DIAG_DELTA 1020 // delta between diagonal neighbours
#define TEST_DELTA 500

// Pins for reading coords
#define X0 2
#define X1 3
#define X2 4
#define Y0 5
#define Y1 6
#define Y2 7

// Intensity levels
#define FULL 1023
#define HALF 516
#define QUARTER 256

// Digital pins to lamp cells
#define RED_OUT 9
#define GREEN_OUT 10
#define YELLOW_OUT 8

// Analog pins to detect faults
#define RED_FAULT A3
#define GREEN_FAULT A2

// Analog pin for reading ambient light
#define AMBIENT A0

// Digital pins to detect movement
#define RED_MOVE 11
#define GREEN_MOVE 12

// Ambient light threshold
```

```
int ambientCmp = VREF;

// Cell address
int address;

// Number of synced cells
int synced = 0;
// Time delta between cells
long deltas[8];
long avgDelta = 0;

// Possible events
enum Event
{
  TIME = 0,
  MOVEMENT = 1,
  PREDICT = 2,
  ADJACENT = 3,
  NOTHING = 4,
  FAIL = 254,
  OK = 255
};

struct Cell
{
  int x;
  int y;
  Event event;
  int target;                // Target intensity
  long start;                // Time at which the LED lit up
  int intensity;             // Brightness intensity
  int sensorPin;             // Pin to read button input
  int faultPin;              // Analog pin to read led fault
  int outPin;                // Digital pin to write led output
  long *neighbourMovements; // Array of neighbour movements (size = 8)
  bool faulty;               // True if the cell is faulty
};

Cell red;
Cell green;

struct Message
{
  byte targetX;
  byte targetY;
  byte sourceX;
  byte sourceY;
  Event event;
```

```
  long time;
};

// Sensor readings
int redSensor;
int greenSensor;
int ambientSensor;

// Neighbour movement memory
long neighbour[8];

int neighbourTranslation[8][2] = {
    {0, 1},
    {1, 1},
    {1, 0},
    {1, -1},
    {0, -1},
    {-1, -1},
    {-1, 0},
    {-1, 1}};

// Number of detected faults
int faults = 0;
long lastTest = 0;

void setup()
{
  // Get start time
  long start = time();

  // Start serial communication
  // Serial.begin(9600);

  // Read cell coordinates
  readCoords();

  // Start I2C communication
  Wire.begin(address);
  Wire.onReceive(receiveMessage);

  // Set pin modes
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(RED_OUT, OUTPUT);
  pinMode(GREEN_OUT, OUTPUT);
  pinMode(YELLOW_OUT, OUTPUT);
  pinMode(RED_MOVE, INPUT_PULLUP);
  pinMode(GREEN_MOVE, INPUT_PULLUP);
  pinMode(X0, INPUT_PULLUP);
  pinMode(X1, INPUT_PULLUP);
  pinMode(X2, INPUT_PULLUP);
```

```
  pinMode(Y0, INPUT_PULLUP);
  pinMode(Y1, INPUT_PULLUP);
  pinMode(Y2, INPUT_PULLUP);

  digitalWrite(LED_BUILTIN, HIGH);
  // Synchronize with other cells
  syncClock(start);
  digitalWrite(LED_BUILTIN, LOW);

  // Turn off LEDs
  digitalWrite(RED_OUT, LOW);
  digitalWrite(GREEN_OUT, LOW);

  // Set up lamp cells
  createCell(&red, RED_MOVE, RED_FAULT, RED_OUT);
  createCell(&green, GREEN_MOVE, GREEN_FAULT, GREEN_OUT);
}

void loop()
{
  // Update cell coordinates
  // readCoords();

  // Read sensors
  readSensors(&red);
  readSensors(&green);

  // Check for faults
  if (time() - lastTest > TEST_DELTA)
  {
    checkFaults(&red);
    checkFaults(&green);
    lastTest = time();
  }
  // Perform actions
  actuate(&red);
  actuate(&green);

  // Blink BUILTIN LED to indicate sync
  if ((time() / 1000) % 2)
    digitalWrite(LED_BUILTIN, HIGH);
  else
    digitalWrite(LED_BUILTIN, LOW);

  digitalWrite(YELLOW_OUT, faults > 0);

  delay(100);
}

void readSensors(Cell *cell)
```

```
{

  // Read sensors
  int movementSensor = digitalRead(cell->sensorPin);
  ambientSensor = analogRead(AMBIENT);

  // Change cell's target intensity
  if (movementSensor == LOW)
  {
    cell->event = MOVEMENT;
    warnNeighbours(cell);
    predictMovement(cell);
  }
  else
    cell->event = NOTHING;

  // Schmitt trigger to detect daytime
  if (ambientSensor < ambientCmp) // cloudy or night time
  {
    ambientCmp = VREF + H / 2;
  }
  else // day time
  {
    ambientSensor = FULL; // Max out ambient light reading
    ambientCmp = VREF - H / 2;
  }
}

void checkFaults(Cell *cell)
{
  // If cell is off, do not check for faults
  if (cell->intensity == 0)
    return;

  // Read fault sensor
  digitalWrite(cell->outPin, HIGH);
  int faultSensor = analogRead(cell->faultPin);
  analogWrite(cell->outPin, cell->intensity);
  // Check if the cell is faulty
  if (faultSensor > 1000 && !cell->faulty)
  {
    cell->faulty = true;
    sendFault(cell);
  }
  else if (faultSensor < 1000 && cell->faulty)
  {
    cell->faulty = false;
    // Serial.print("Fault fixed ");
    // Serial.print(cell->x);
    // Serial.print(",");
```

```
    // Serial.println(cell->y);
    sendFault(cell);
  }
}

void readCoords()
{
  // Read DIP switch and update cell coordinates
  red.x = digitalRead(X0) ^ 1 | (digitalRead(X1) ^ 1) << 1 |
    (digitalRead(X2) ^ 1) << 2;
  red.y = digitalRead(Y0) ^ 1 | (digitalRead(Y1) ^ 1) << 1 |
    (digitalRead(Y2) ^ 1) << 2;

  green.x = red.x + 1;
  green.y = red.y;

  address = address(red.x, red.y);
}

void syncClock(long startTime)
{
  // Initialize 4 neighbour coordinates
  int neighbours[4][2] = {
      {0, 1},
      {1, 0},
      {0, -1},
      {-1, 0}};

  delay(2500);

  // Send message from red cell (x,y) and green cell (x+1, y)
  for (int i = 0; i < 2; i++)
  {
    // Send message to all 4 neighbours of current cell
    for (int i = 0; i < 4; i++)
    {
      Message message;
      message.targetX = red.x + i + neighbours[i][0];
      message.targetY = red.y + neighbours[i][1];
      message.sourceX = red.x + i;
      message.sourceY = red.y;
      message.event = TIME;
      message.time = time() / 10;

      // Send message
      sendMessage(message);
    }
  }

  //  Wait for messages from all neighbours
```

```
  while (millis() - startTime < 5000 and synced < 8)
    ;

  // calculate average time delta
  for (int i = 0; i < synced; i++)
  {
    avgDelta += deltas[i];
  }
  if (synced)
    avgDelta /= synced;
  // Adjust delta from 10ms unit to 1ms unit
  avgDelta *= 10;
  // Serial.print("Delta: ");
  // Serial.println(avgDelta);
}

void actuate(Cell *cell)
{
  // Do red event
  switch (cell->event)
  {
  case PREDICT:
  case MOVEMENT:
    // Send to other devices
    cell->start = time();
    cell->target = FULL;
    break;

  case ADJACENT:
    // If lamp is already at full brightness, ignore lower levels
    if (cell->target == FULL)
      break;

    cell->start = time();
    cell->target = HALF;
    break;
  default:
    break;
  }

  // Check if target duration hasnt been reached, target=QUARTER if not
  long elapsed = time() - cell->start;
  cell->target = elapsed < DURATION ? cell->target : QUARTER;

  // Adjust brightness according to ambient light
  cell->intensity = cell->target - ambientSensor;
  cell->intensity = constrain(cell->intensity, 0, 1024);
  cell->intensity = map(cell->intensity, 0, 1024, 0, 256);
```

```

  analogWrite(cell->outPin, cell->intensity);
}

void sendMessage(Message msg)
{
  if ((msg.targetX == red.x && msg.targetY == red.y) ||
      (msg.targetX == green.x && msg.targetY == green.y))
  {
    handleMessage(msg);
    return;
  }
  else if (msg.targetX == 255 || msg.targetY == 255) //
     Message out of bounds
  {
    return;
  }

  // Send message over I2C
  Wire.beginTransmission(address(msg.targetX, msg.targetY));
  Wire.write(msg.targetX << 4 | msg.targetY);
  Wire.write(msg.sourceX << 4 | msg.sourceY);
  Wire.write(msg.event);
  Wire.write(msg.time >> 24);
  Wire.write(msg.time >> 16);
  Wire.write(msg.time >> 8);
  Wire.write(msg.time);
  Wire.endTransmission();
}

void handleMessage(Message msg)
{
  switch (msg.event)
  {

  case TIME:
  {
    // Update synced
    deltas[synced] = millis() / 10 - msg.time;
    synced++;
    break;
  }
  case MOVEMENT:
  {
    Cell *cell = getTargetCell(msg);
    cell->event = ADJACENT;
    actuate(cell);
    saveNeighbourMovement(msg);
    break;
  }
```

```
  case PREDICT:
  {
    Cell *cell = getTargetCell(msg);
    cell->event = PREDICT;
    actuate(cell);
    saveNeighbourMovement(msg);
    break;
  }
  case FAIL:
  {
    if (red.x == 0 && red.y == 0)
      faults++;
    break;
  }
  case OK:
  {
    if (red.x == 0 && red.y == 0 && faults > 0)
      faults--;
    break;
  }
  default:
    break;
  }
}

void sendFault(Cell *cell)
{
  // Send message to main cell (0,0)
  Message message;
  message.targetX = 0;
  message.targetY = 0;
  message.sourceX = cell->x;
  message.sourceY = cell->y;
  message.event = cell->faulty ? FAIL : OK;
  message.time = time() / 10;

  // Send message
  sendMessage(message);
}

void predictMovement(Cell *cell)
{
  long currTime = time() / 10;
  int nextNeighbour = -1;
  // Check if neighbours have moved in the last 5 seconds
  for (int i = 0; i < 8; i++)
  {
    // Check if neighbour has any movement detected
    if (cell->neighbourMovements[i])
      // Check if neighbour has moved recently (in 10 ms)
```

```
      if ((i % 2 && currTime - cell->neighbourMovements[i] < DIAG_DELTA) ||
          (currTime - cell->neighbourMovements[i] < CARD_DELTA))
      {
        // If there is movement detected, send a message to next neighbour
        nextNeighbour = (i + 4) % 8;
        sendPrediction(cell, nextNeighbour);
      }
  }
}

void sendPrediction(Cell *cell, int neighbour)
{
  // Send message to next neighbour
  Message message;
  message.targetX = cell->x + neighbourTranslation[neighbour][0];
  message.targetY = cell->y + neighbourTranslation[neighbour][1];
  message.sourceX = cell->x;
  message.sourceY = cell->y;
  message.event = PREDICT;
  message.time = time() / 10;

  sendMessage(message);
}

void saveNeighbourMovement(Message msg)
{
  // Save movement
  int index = getNeighbourIndex(msg.sourceX - msg.targetX, msg.sourceY - msg.targetY);
  if (index != -1)
    getTargetCell(msg)->neighbourMovements[index] = msg.time;
}

int getNeighbourIndex(int x, int y)
{
  // Get neighbour index from neighbourTranslation array
  for (int i = 0; i < 8; i++)
  {
    if (x == neighbourTranslation[i][0] && y == neighbourTranslation[i][1])
      return i;
  }
  return -1;
}
```

```
Cell *getTargetCell(Message msg)
{
  // Return red cell if target is red
  return (msg.targetX == red.x && msg.targetY == red.y) ? &
    red : &green;
}

void receiveMessage(int numBytes)
{
  Message msg;
  while (Wire.available() > 0)
  {
    // Read 8 bits for target cell coordinates
    int cell = Wire.read();
    msg.targetX = cell >> 4;
    msg.targetY = cell & 0x0F;

    // Read 8 bit for source
    int source = Wire.read();
    msg.sourceX = source >> 4;
    msg.sourceY = source & 0x0F;
    // Read 8 bit for event
    msg.event = (Event)Wire.read();
    // Read 32 bit for time
    msg.time = (long)Wire.read() << 24 | (long)Wire.read() <<
     16 | (long)Wire.read() << 8 | (long)Wire.read();
  }

  handleMessage(msg);
}

void warnNeighbours(Cell *cell)
{
  // Initialize 8 neighbour coordinates
  int neighbours[8][2] = {
      {0, 1},
      {1, 0},
      {0, -1},
      {-1, 0},
      {1, 1},
      {1, -1},
      {-1, 1},
      {-1, -1}};

  // Send message to all 8 neighbours of current cell
  for (int i = 0; i < 8; i++)
  {
    Message message;
    message.targetX = cell->x + neighbours[i][0];
    message.targetY = cell->y + neighbours[i][1];
```

```
    message.sourceX = cell->x;
    message.sourceY = cell->y;
    message.event = MOVEMENT;
    message.time = time() / 10;

    // Send message
    sendMessage(message);
  }
}

void createCell(Cell *cell, int sensorPin, int faultPin, int
    outPin)
{
  // Initialize cell with correct values
  cell->sensorPin = sensorPin;
  cell->faultPin = faultPin;
  cell->outPin = outPin;
  cell->neighbourMovements = (long *)calloc(8, sizeof(long));
  cell->faulty = false;
}
```

Listing 1.1. Arduino controller code