Water refilling system

Introducere


```
Nume: Popovici Robert-Adrian
```

```
Grupa: 335CA
```


Obiectivul acestui proiect consta in realizarea unui sistem automat de umplere a diverselor recipiente cu lichide. Inregul sistem este centrat in jurul unei placute Arduino Mega 2560, aceasta fiind responsabila pentru controlul pieselor esentiale ale acestui proiect (pompa de apa, releul care alimenteaza pompa, etc.), dar si a componentelor care care servesc drept input si output pentru utilizatori (matricea de leduri, potentiomentru liniar, etc.).

Utilitatea proiectului este destul de evidenta si poate fi rezumata prin urmatoarele idei:

- automatizarea unei sarcini uzuale si destul de frecvente
- oferirea unei precizii mult mai bune si micsorarea duratei de asteptare
- evitarea diverselor accidente care pot fi cauzate de erorile umane

Descriere generală

Interactiunea utilizatorului cu intregul sistem se realizeaza prin intermediul unui potentiomentru liniar, acesta avand rolul de a seta volumul de lichind care urmeaza sa fie turnat in recipient. In timp real aceast volum va fi afisata de catre Arduino pe matricea led 8×8 in format decimal, controlul acesteia fiind realizat pur software (fara a folosi niciun driver auxiliar de control).

Dupa ce volumul de lichid a fost stabilit utilizatorul va da comanda de start prin apasarea unui push-button. La apasarea butonului de start, modulul Arduino va activa releul care la randul lui va pune in functiune pompa de apa. Pentru a calcula timpul necesar de functionare al pompei vom presupune ca aceasta are un debit aproximativ constant, pe care il vom calcula in prealabil.

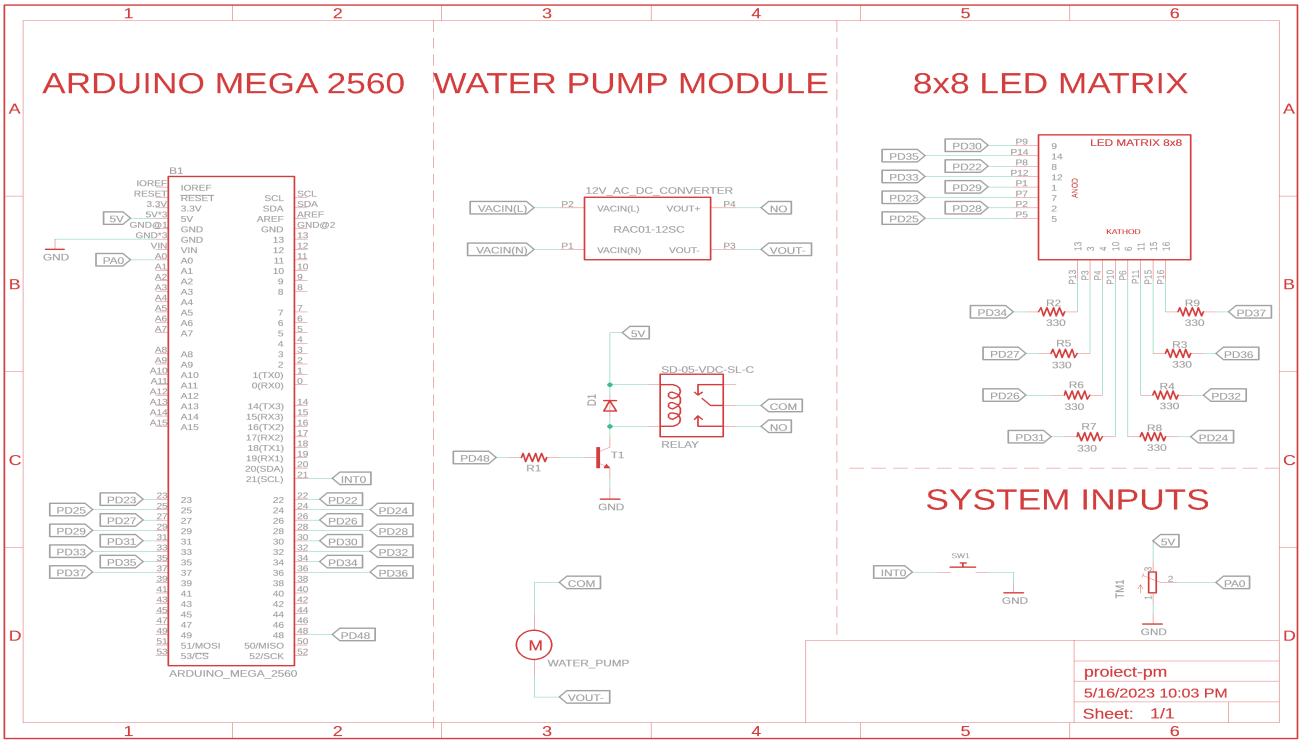
Dupa scurgerea timpului determinat anterior, pompa se va opri, iar sistemul este pregatit pentru a procesa noi comenzi.

Schema bloc

Hardware Design

Piesele folosite in cadrul proiectului sunt:

- Arduino Mega 2560
- Pompa de apa submersibila 12V
- Matrice de leduri 8×8
- Convertor AC-DC 12V
- Releu 5V
- Rezistente 330 ohm
- Potentiometru liniar 10k ohm
- Push button
- Fire dupont



Software Design

Organizare

Codul este structurat in mai multe fisiere auxiliare grupate in functie de logica pe care o indeplinesc. Astfel vom prezenta pe scurt principalele module ale aplicatiei, urmand sa analizam in detaliu modul lor de functionare in sectiuniile urmatoare.

Button

Clasa care permite interfatarea unui buton. Aceasta are incorporat un modul de debouncer si poate fi configurata pentru a declansa diverse actiuni in functie de modul in care apasam butonul.

Componente:

- button.h
- buttonConfig.h
- buttonConfig.cpp

Clock1ms

Timer cu frecventa de 1kHz, folosit de majoritatea modulelor aplicatiei.

Componente:

- clock1ms.h
- clock1ms.cpp
- clockConfig.h
- clockConfig.cpp

EventHandler

Clasa generica care permite declansare unor actiuni la un anumit moment de timp aleator, ales de apelant.

Componente:

- eventHandler.h

LedMatrix

Driver folosit pentru a controla matricea de leduri. Acesta poate fi configurat la modul general pentru a controla orice matrice de leduri (indiferent de dimensiune) sau chiar o grupare de mai multe matrici.

Componente:

- ledMatrix.h
- ledMatrixConfig.h
- ledMatrixConfig.cpp

Logger

Principalul modul folosit pentru a afisa mesaje cu rol informativ legate de starea sistemului.

Componente:

- logger.h
- logger.cpp

PinInfo

Wrapper peste un pin al Arduino. Controleaza un pin prin intermediul registrelor specifice (DDR, PIN, PORT) si expune o interfata simplu de utilizat pentru a configura pinul in diverse moduri.

Componente:

- pinInfo.h

WaterPump

Clasa folosita pentru a controla pompa de apa.

Componente:

- waterPump.h
- waterPump.cpp
- waterPumpConfig.h
- waterPumpConfig.cpp

Program Flow

Fisierul **main.ino** este responsabil pentru intializarea tuturor componentelor si actualizarea periodica a acestora.

```
#include "config.h"

static constexpr int BAUD_RATE = 9600;

void setup() {
        Logger::setup(BAUD_RATE);

        setupClock();

        setupLedMatrix();

        setupButton();
```

```
        setupWaterPump();

        setupEventHandler();
}

void loop() {
        if (clock_1ms::periodElapsed()) {

                updatePotentiometer();

                updateEvents();

                updateButton();
        }

        updateLedMatrix();
}
```

Modulul **clock1ms.cpp** configureaza intern o intrerupere care va fi declansata o data la 1ms. In cadrul intreruperii vom incrementa valoarea tickCounter, folosita pentru a simula manual operatiile de delay().

```
#include "clock1ms.h"

static volatile long long tickCounter = 0LL;

ISR(TIMER1_COMPA_vect) {
        tickCounter++;
}

namespace clock_1ms {
        void startClock() {
                // disable all interrupts
                cli();

                // we want to check status every 1ms (f = 1KHz)
                // f_clk = 16MHz => N = 64 and OCR1A = 249

                TCNT1 = 0;
                TCCR1A = 0;
                TCCR1B = 0;

                OCR1A = 249;
                TCCR1B |= (1 << WGM12);   // CTC mode
                TCCR1B |= (1 << CS11);    // 64 prescaler
                TCCR1B |= (1 << CS10);

                TIMSK1 |= (1 << OCIE1A);  // OCR1A compare match interrupt

                // enable back interrupts
                sei();
```

```
        }

        bool periodElapsed() {
                const long long elapsed = tickCounter - lastTickCounter;
                lastTickCounter = tickCounter;
                return elapsed > 0;
        }

        long long getTickCounter() {
                return tickCounter;
        }
}
```

Modulul button este alcatuit din 2 parti:

- In **button.h** avem implementarea efectiva a butonului cu modul intern de debouncer. Aceasta primeste un parametru generic de tipul callback, care va fi apelat, in functie de configurarea aleasa, fie atunci cand apasam butonul, fie cand il eliberam. Pentru aplicatia vom folosi modul (FALLING_EDGE), deoarece ne intereseaza sa pornim pompa o data ce utilizatorul a apasat butonul.

```
#pragma once

#include "clock1ms.h"
#include "pinInfo.h"

class AbstractButton {
public:
        AbstractButton(PinInfo pinInfo) {
                this->pinInfo = pinInfo;
        }

protected:
        PinInfo pinInfo;
};

class AbstractCallableButton : public AbstractButton {
public:
        enum class CallbackActivation {
                RISING_EDGE,
                FALLING_EDGE
        };

        AbstractCallableButton(PinInfo pinInfo) : AbstractButton(pinInfo) {}

        static inline CallbackActivation getFallingEdge() {
                return CallbackActivation::FALLING_EDGE;
        }

        static inline CallbackActivation getRisingEdge() {
                return CallbackActivation::RISING_EDGE;
        }
```

```
        virtual void checkButton() = 0;
};

template<typename T>
class BaseCallableButton : public AbstractCallableButton {
public:
        BaseCallableButton(PinInfo pinInfo, const T& callback,
CallbackActivation activationEdge)
                : AbstractCallableButton(pinInfo),
                  callback(callback) {

                this->activationEdge = activationEdge;
                if (activationEdge == CallbackActivation::RISING_EDGE) {
                        pinInfo.configure(PinInfo::PinType::_INPUT_PULLDOWN);
                } else {
                        pinInfo.configure(PinInfo::PinType::_INPUT_PULLUP);
                }
        }

        inline bool isActiveOnFallingEdge() {
                return activationEdge == CallbackActivation::FALLING_EDGE;
        }

        inline bool isActiveOnRisingEdge() {
                return activationEdge == CallbackActivation::RISING_EDGE;
        }

protected:
        CallbackActivation activationEdge;
        const T& callback;
};

template<typename T>
class CallableButton : public BaseCallableButton<T> {
public:
        using CallbackActivation = AbstractCallableButton::
CallbackActivation;

        CallableButton(PinInfo pinInfo, const T& callback,
CallbackActivation activationEdge)
                : BaseCallableButton<T>(pinInfo, callback, activationEdge) {
}

        CallableButton(PinInfo pin, const T& callback)
                : CallableButton<T>(pin, callback, AbstractCallableButton::
getFallingEdge()) {}

        void checkButton() override {
                const long long tickCounter = clock_1ms::getTickCounter();
```

```
                int currentButtonState = this->pinInfo.read();
                if (lastButtonState != currentButtonState) {
                        lastDebounceTime = tickCounter;
                }

                if (tickCounter - lastDebounceTime > DEBOUNCE_TIME_MILLIS) {
                        if (currentButtonState != debouncedButtonState) {
                                debouncedButtonState = currentButtonState;
                                if (debouncedButtonState == LOW &&
isActiveOnFallingEdge() ||
                                    debouncedButtonState == HIGH &&
isActiveOnRisingEdge()) {
                                        callback();
                                }
                        }
                }

                lastButtonState = currentButtonState;
        }

private:
        static constexpr long long DEBOUNCE_TIME_MILLIS = 50;

        int lastButtonState;
        int debouncedButtonState;
        long long lastDebounceTime;
};
```

- In **buttonConfig.cpp** vom instantia butonul si vom configura actiunile declansate atunci cand aceasta este apasat. Principala actiune va fi pornirea pompei de apa, daca aceasta nu era deja in starea RUNNING. Dupa aceea, vom calcula timpul necesar de umplere si vom insera un nou eveniment in event handler pentru a opri pompa dupa timpul calculat anterior.

```
#include "buttonConfig.h"

static constexpr PinInfo BUTTON_PIN_INFO = {&DDRD, &PIND, &PORTD, PD0};
AbstractCallableButton* button;

void setupButton() {
        auto buttonCallback = []() {
                if (waterPump.getState() == WaterPump::WorkingState::RUNNING)
                        return;

                // multiply displayNumber with SCALE_FACTOR for better
granularity
                static constexpr int SCALE_FACTOR = 10;
                long long fillTime = WaterPump::getFillTime(displayNumber *
SCALE_FACTOR);
                long long currentTime = clock_1ms::getTickCounter();
                long long expireTime = fillTime + currentTime;
```

```
                auto& logger = Logger::getLogger();
                logger.debug().append("Current time: ").append(currentTime).
commit();
                logger.debug().append("Fill time: ").append(fillTime).commit
();
                logger.debug().append("Expire time: ").append(expireTime).
commit();

                logger.info().append("Starting pump...").commit();
                waterPump.startPump();

                auto eventCallback = []() {
                        auto& logger = Logger::getLogger();
                        logger.info().append("Stopping pump: ").append(
clock_1ms::getTickCounter()).commit();
                        waterPump.stopPump();
                };

                using event_callback_t = decltype(eventCallback);
                eventHandler.addEvent<event_callback_t>(expireTime,
eventCallback);
        };


        using button_callback_t = decltype(buttonCallback);
        button = new CallableButton<button_callback_t>(BUTTON_PIN_INFO,
buttonCallback);
}

void updateButton() {
        button->checkButton();
}
```

Modulul ledmatrix este alcatui in mod simlar din 2 parti:

- In **ledMatrixConfig.cpp** vom initializa driver-ul pentru matricea de leduri. Acesta primeste ca argumente in constructor doi vectori de tipul PinInfo. Primul vector reprezinta pinii care vor controla randurile matricei de leduri, iar al doilea vector reprezinta pinii de pe coloane. Functia updateLedMatrix este folosita pentru a afisa valoarea tensiunii de pe potentiometru, dupa ce o convertim la o valoarea normalizata. Dupa cum se poate vedea implementarea este destul de generica si nu tine cont de dimensiunea matricei

```
#include "ledMatrixConfig.h"

static const std::array<PinInfo, LED_MATRIX_ROWS> ROW_PINS =
        {{&DDRC, &PINC, &PORTC, PC7},
         {&DDRC, &PINC, &PORTC, PC2},
         {&DDRA, &PINA, &PORTA, PA0},
         {&DDRC, &PINC, &PORTC, PC4},
         {&DDRA, &PINA, &PORTA, PA7},
         {&DDRA, &PINA, &PORTA, PA1},
```

```
        {&DDRA, &PINA, &PORTA, PA6},
        {&DDRA, &PINA, &PORTA, PA3}};

static const std::array<PinInfo, LED_MATRIX_COLS> COL_PINS =
       {{&DDRC, &PINC, &PORTC, PC3},
        {&DDRA, &PINA, &PORTA, PA5},
        {&DDRA, &PINA, &PORTA, PA4},
        {&DDRC, &PINC, &PORTC, PC6},
        {&DDRA, &PINA, &PORTA, PA2},
        {&DDRC, &PINC, &PORTC, PC5},
        {&DDRC, &PINC, &PORTC, PC1},
        {&DDRC, &PINC, &PORTC, PC0}};


LedMatrixDriver<LED_MATRIX_ROWS, LED_MATRIX_COLS> ledMatrix(ROW_PINS,
COL_PINS);

void setupLedMatrix() {
       // ToDo: add setup if necessary
}

void updateLedMatrix() {
       displayNumber = map(potentiometerValue, 0, 1023, DISPLAY_RANGE_HIGH,
DISPLAY_RANGE_LOW);
       ledMatrix.drawNumber(displayNumber, DISPLAY_OFFSET_ROW,
DISPLAY_OFFSET_COL);
}
```

- In **ledMatrix.h** avem implementarea efectiva a driver-ului. In functie de sablonul pe care vrem sa-l afisam vom aprinde/stinge pe rand ledurile din matrice, iar ochiul uman va percepe o iluminare continua datorita frecventei ridicate.

```
template<size_t _RowSize, size_t _ColSize>
class LedMatrix {
private:
       void setUpLedMatrix() {
              // set row pins as OUTPUT
              for (auto pin : this->getRowPins()) {
                     pin.configure(PinInfo::PinType::_OUTPUT);
              }

              // set column pins as OUTPUT
              for (auto pin : this->getColPins()) {
                     pin.configure(PinInfo::PinType::_OUTPUT);
              }

              // turn off all matrix leds
              this->turnAllOff();
       }

       inline void write(PinInfo pin, int value) {
```

```
                pin.write(value);
        }

public:
        LedMatrix(const std::array<PinInfo, _RowSize>& rowPins, const std::
array<PinInfo, _ColSize>& colPins)
                : rowPins(rowPins),
                  colPins(colPins) {

                this->setUpLedMatrix();
        }

        void turnOnLed(int row, int col) {
                for (int i = 0; i < _ColSize; i++) {
                        if (i != col) {
                                write(colPins[i], HIGH);
                        } else {
                                write(colPins[i], LOW);
                        }
                }
                write(rowPins[row], HIGH);
        }

        void turnOffLed(int row, int col) {
                write(rowPins[row], LOW);
                for (int i = 0; i < _ColSize; i++) {
                        write(colPins[i], LOW);
                }
        }

        void turnAllOff() {
                for (auto pin : rowPins) {
                        write(pin, LOW);
                }

                for (auto pin : colPins) {
                        write(pin, LOW);
                }
        }

        const std::array<PinInfo, _RowSize>& getRowPins() {
                return rowPins;
        }

        const std::array<PinInfo, _ColSize>& getColPins() {
                return colPins;
        }

private:
        std::array<PinInfo, _RowSize> rowPins;
        std::array<PinInfo, _ColSize> colPins;
```

```
};


template<size_t _RowSize, size_t _ColSize>
class LedMatrixDriver : public LedMatrix<_RowSize, _ColSize> {
public:
        LedMatrixDriver(const std::array<PinInfo, _RowSize>& rowPins, const
std::array<PinInfo, _ColSize>& colPins)
                : LedMatrix<_RowSize, _ColSize>(rowPins, colPins) {}

        void drawDigit(int digit, int offsetRow, int offsetCol) {
                for (int r = 0; r < DIGIT_ROWS; r++) {
                        for (int c = 0; c < DIGIT_COLS; c++) {
                                int row = r + offsetRow;
                                int col = c + offsetCol;
                                if (digits[digit][r][c] == '1') {
                                        this->turnOnLed(row, col);
                                        this->turnOffLed(row, col);
                                }
                        }
                }
        }

        void drawDigit(int digit) {
                drawDigit(digit, 0, 0);
        }

        void drawNumber(int number, int offsetRow, int offsetCol) {
                int firstDigit = number / 10;
                int secondDigit = number % 10;
                drawDigit(firstDigit, offsetRow, offsetCol);
               drawDigit(secondDigit, offsetRow, offsetCol + DIGIT_COLS + 1);
        }

        void drawNumber(int number) {
                drawNumber(number, 0, 0);
        }
};
```

Modulul waterpump este folosit pentru a interactiona cu pompa de apa. Acesta expune doua metode startPump si stopPump pentru a porni respectiv opri pompa.

```
#include "waterPump.h"

WaterPump::WaterPump(PinInfo pinInfo) {
        this->pinInfo = pinInfo;
        pinInfo.configure(PinInfo::PinType::_OUTPUT);
        stopPump();
}
```

```
void WaterPump::startPump() {
        if (state == WorkingState::RUNNING) {
                return;
        }
        state = WorkingState::RUNNING;
        pinInfo.write(LOW);
}

void WaterPump::stopPump() {
        pinInfo.write(HIGH);
        state = WorkingState::NOT_RUNNING;
}
```

Fisierul **eventHandler.h** contine implementarea message bus-ului folosit pentru a opri pompa o data ce timpul de umplere s-a scurs. Acesta retine intern un set cu toate evenimentele care trebuie planificate sortate dupa timpul de expirare. Atunci cand apelam functia addEvent vom insera un nou eveniment in set, iar la apelarea functiei checkEvents vom scoate toate evenimentele care au expirat ruland in prealabil functiile de callback asociate lor. In implementarea curenta avem cel mult un eveniment activ in set, deoarece ignoram actiunile de apasarea ale butonului atunci cand este deja in functiune. Datorita acestui fapt memoria heap consumata este destul de mica.

```
#pragma once

#include <ArduinoSTL.h>
#include <set>

#include "clock1ms.h"

class EventHandler {
public:
        template<typename T>
        void addEvent(long long expireTime, const T& callback) {
                static long long lastEventId = 0;
                BaseEvent* event = new Event<T>(lastEventId, expireTime,
callback);
                lastEventId++;
                eventQueue.insert(event);
        }

        void checkEvents() {
                long long tickCounter = clock_1ms::getTickCounter();
                while (!eventQueue.empty()) {
                        BaseEvent* event = *eventQueue.begin();
                        if (event->expireTime <= tickCounter) {
                                eventQueue.erase(eventQueue.begin());
                                event->call();
                                delete event;
                        } else {
                                break;
                        }
                }
```

```
        }

private:
        struct BaseEvent {
                long long eventId;
                long long expireTime;

                BaseEvent(long long eventId, long long expireTime) {
                        this->eventId = eventId;
                        this->expireTime = expireTime;
                }

                bool operator<(const BaseEvent& rhs) const {
                        if (expireTime == rhs.expireTime)
                                return eventId < rhs.eventId;
                        return expireTime < rhs.expireTime;
                }

                virtual void call() {
                        // can't make it pure virtual because we need to use
less than operator inside set comparator
                }
        };

        template<typename T>
        struct Event : public BaseEvent {
                const T& callback;

                Event(long long eventId, long long expireTime, const T&
callback)
                        : BaseEvent(eventId, expireTime),
                          callback(callback) {}

                void call() override {
                        callback();
                }
        };

private:

        struct EventComparator {
                bool operator()(const BaseEvent* e1, const BaseEvent* e2)
const {
                        return *e1 < *e2;
                }
        };

        std::set<BaseEvent*, EventComparator> eventQueue;
};
```

Fisierul **pinInfo.h** reprezinta un wrapper peste un pin al Arduino. Acesta primeste ca parametrii in constructor pointeri catre registrele DDR, PIN si PORT alea pin-ului si le foloseste mai apoi pentru a scrie valori logice pe pin, sau pentru a configura pin-ul in diverse moduri (INPUT, OUTPUT, INPUT_PULLUP sau INPUT_PULLDOWN).

```
#pragma once

#include <Arduino.h>

struct PinInfo {
        enum class PinType {
                _INPUT,
                _OUTPUT,
                _INPUT_PULLUP,
                _INPUT_PULLDOWN
        };

        volatile uint8_t* DDR;
        volatile uint8_t* PIN;
        volatile uint8_t* PORT;
        uint8_t ID;

        inline volatile uint8_t& getDDR() {
                return *DDR;
        }

        inline volatile uint8_t& getPIN() {
                return *PIN;
        }

        inline volatile uint8_t& getPORT() {
                return *PORT;
        }

        inline uint8_t getID() {
                return ID;
        }

        inline int read() {
                return getPIN() & (1 << getID());
        }

        inline void write(int value) {
                if (value == LOW) {
                        getPORT() &= ~(1 << getID());
                } else {
                        getPORT() |= (1 << getID());
                }
        }

        inline int configure(PinType type) {
```

```
                if (type == PinType::_INPUT) {
                        getDDR() &= ~(1 << getID());
                } else if (type == PinType::_OUTPUT) {
                        getDDR() |= (1 << getID());
                } else if (type == PinType::_INPUT_PULLUP) {
                        getDDR() &= ~(1 << getID());
                        write(HIGH);
                } else if (type == PinType::_INPUT_PULLDOWN) {
                        getDDR() &= ~(1 << getID());
                        write(LOW);
                }
        }
};
```

Biblioteci

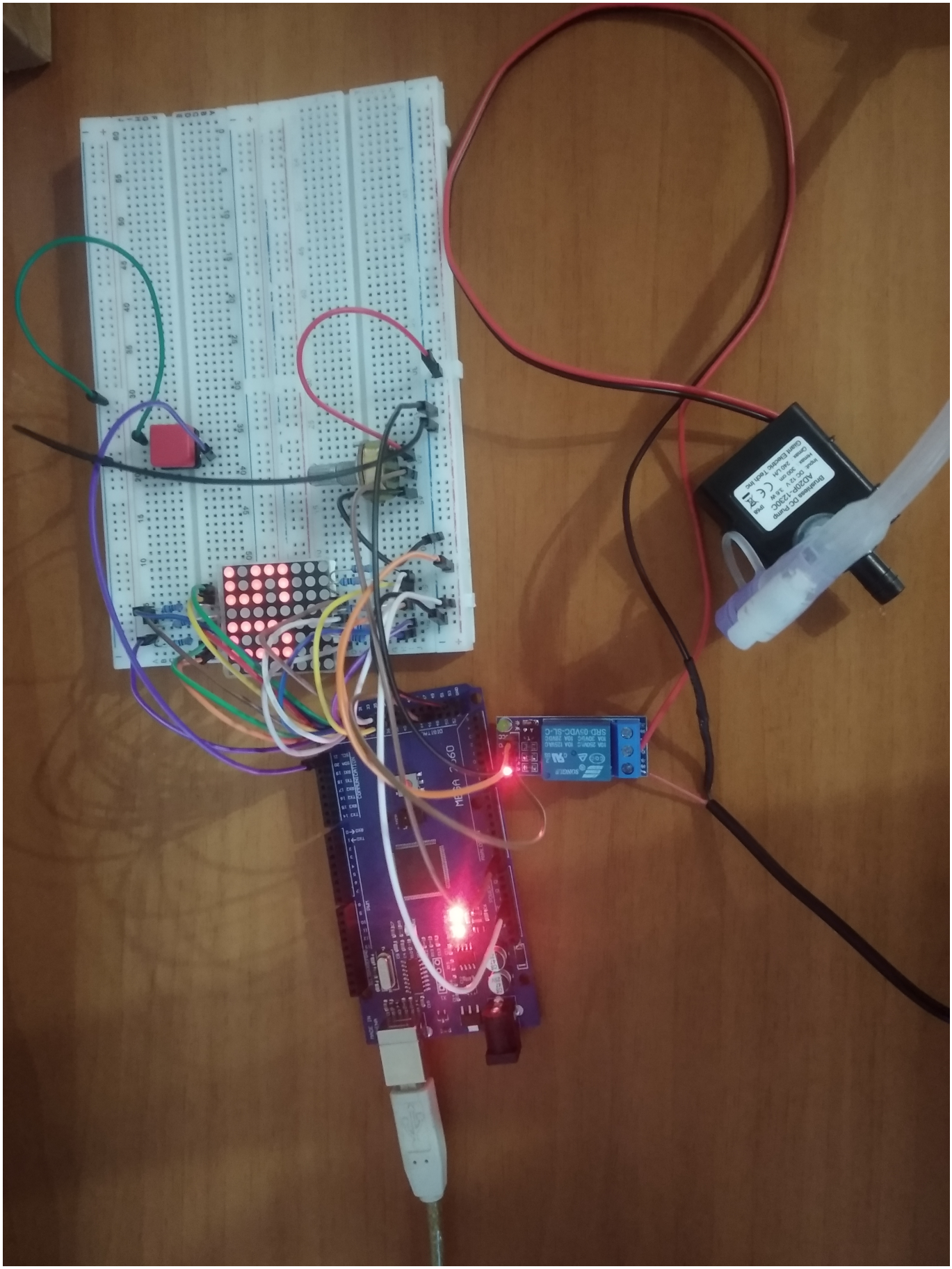
- ArduinoSTL

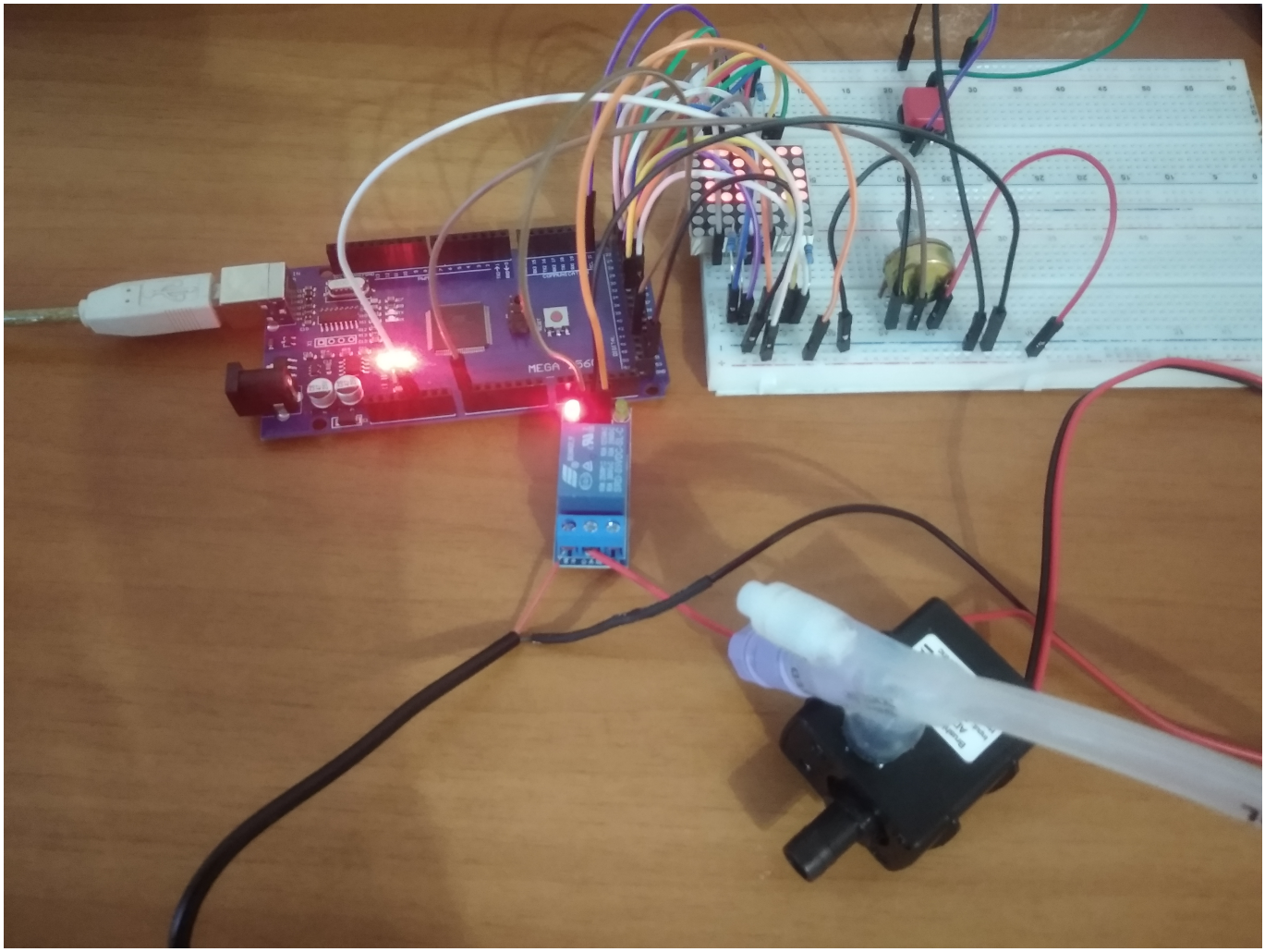
Mediu de dezvoltare

Mediul de dezvoltare folosit in realizare proiectului: **Arduino IDE**

Rezultate Obţinute

Proiect final


Brushless DC Pump
SRD-05VDC-SL-C



Demo

Mai multe poze cu circuitul si un cateva filmulete cu scop demonstrativ se pot gasi la acest link

Concluzii

Proiectul a fost foarte interesant de realizat atat din punct de vedere hardware cat si software, iar timpul pe care l-am petrecut lucrand la acesta mi-a permis sa aprofundez mult mai bine conceptele prezentate la laborator si totodata mi-a oferit sansa sa imi exersez skill-urile de software design. Majoritatea cerintelor functionale pe care le-am mentionat in sectiunea de introducere a proiectului au fost indeplinite cu succes, iar overall sunt multumit de rezultatele obtinute.

Singurul aspect care ar putea fi imbunatatit este momentul in care oprim pompa de apa. In varianta actuala am presupus ca aceasta are un debit relativ constant si pe baza acestuia am calculat timpul necesar de umplere. In realitate debitul nu este chiar constant, acesta fiind mai mic la inceput si dupa aceea crescand treptat pana ce pompa ajunge in modul optim de functionare. Din aceasta cauza, proiectul are o acuratete mai buna pentru volume mai mici de lichid, iar pentru volume mari exista un surplus introdus in recipient. Rezolvarea acestei probleme este relativ simpla prin adaugarea unui

senzor de greutate care va monitoriza in permanenta canitatea de lichid care se afla in vas.

Download

Intregul cod este disponibil la acest link sau sub forma de arhiva zip: water_refilling_system.zip

Jurnal

- 24.04.2023 → Alegere tema proiect
- 01.05.2023 → Comanda piese
- 04.05.2023 → Prima incercare de a programa matricea de leduri
- 05.05.2023 → Citire date de pe potentiometru + afisare de cifre pe matricea de leduri
- 06.05.2023 → Implementare timer cu intreruperi
- 14.05.2023 → Hardware finalizat + restructurare minora a codului
- 21.05.2023 → Adaugarea tuturor partilor de cod lipsa + impartirea in module
- 23.05.2023 → Software finalizat
- 29.05.2023 → Finalizare pagina wiki

Bibliografie/Resurse

Resure Hardware

- https://www.electronicshub.org/arduino-mega-pinout/
- https://ocw.cs.pub.ro/courses/_media/pm/doc8272.pdf
- https://www.youtube.com/watch?v=58XWVDnB7Ss&ab_channel=Robojax
- https://arduinogetstarted.com/tutorials/arduino-controls-pump
- https://www.circuitstoday.com/interfacing-8x8-led-matrix-with-arduino

Resure Software

- https://github.com/mike-matera/ArduinoSTL
- https://ocw.cs.pub.ro/courses/pm/lab/lab3-2023