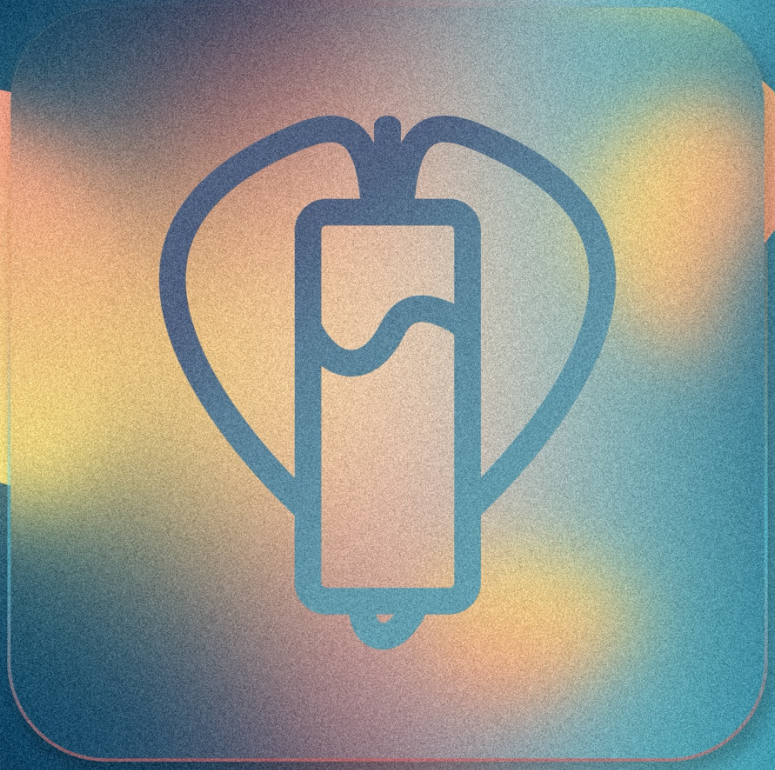
PUMP DEMO

[embedded]



IBM

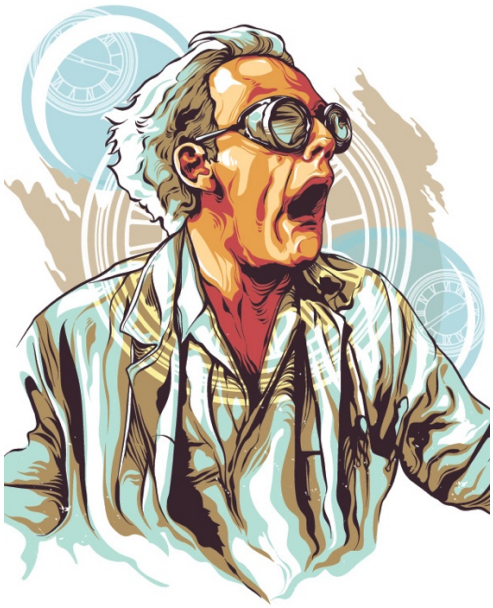
NACKADEMIN

"Roads? Where We're Going, We Don't Need Roads."
- Dr. Emmett Brown.

Foreword

In this project we have been using two arduino nanos, one for lights(B) and one for controlling the pump(A). The ring lights on the buttons are still on the control arduino for convenience. Everything is written In c++ but with use of the Arduino.h library, so it's not truly c++. There are some tech debt that I will list for future fixes, but since time is of the essence I will leave that for you; the hero whom has been chosen to carry the torch on this journey of arcane knowledge paved by thy elders.

If you have any questions feel free to reach me at LinkedIn



-Good luck
Robin.

Controll Arduino code (Arduino A):

Functions:

buttonXIsTriggered();

when a button is pushed, it sets the triggered state to true,
as an extra check before the debounce() function.

debounce();

check if the button is actually pressed with the buttonAisTriggered()
function, den it checks if it was x millis long ago. If so, it calls the function.

pumpcontroller();

checks if the pump is on or off, and sets it to the opposite.

pumpSerial();

Reads incomming data as String, converts It to char and counts amount of characters.
If the input is more than one character it's a potentiometer value.
And send the incomming data to the setPot(cmd); function.
Else it will send it to the command handler, commandHandler(buf);

commandHandler(char comdata);

The incomming char passed from pumpSerial(); runs through a switch case, to trigger the right function.

Eg.

```
void commandHandler(int comdata) {

  switch (comdata) {

    case 'A': // A Starts pump
      pumpcontroller('A'); //Pumpcontroller in
pumpOnOff.h
      break;

    case 'B'://B stops pump
      pumpcontroller('B');//Pumpcontroller in
pumpOnOff.h

      break;
  }

}
```

void setPot(String potVal);

Gets string passed from pumpSerial();
Checks if string starts with any of the 3 letters (X,Y,Z) that represent potentiometers 1,2&3.
Then takes the value that comes after the letter. Eg: X166 sets potentiometer 1 value to 166.

```
if (potVal.startsWith("X")) {
  int input = potVal.substring(3).toInt();

  if (input > 1 && input < 255) {
    pump_speed = input;
  }
```

send_json();
In json.h. Takes sensor readings and pump id, and turns it into a nice json object. And Serial.prints it.

tempReading();
Reads temperature and stores it in temperature variable.
Checks if old is different from new value. If true, call send_json();

runTime();
To determine how long the arduino has been running.
Breaks down millis() into a seconds counter which is NOT affected by rollovers .
Then hours/ minutes/seconds formatted as: "%02d:%02d:%02d", saved to variable
`char runTimeBuffer[21];` in declare.h

```
Runtime00:00:45
```

Internal time counting using the on-board oscillator/or ceramic resonator as a clocking source may be wrong up to 0.8%.
So counting 1000 seconds with internal timing might be something between 992 and 1008 seconds, actually.

This is no longer in use, but saved since theres still a debate if it should be in use later on:

void safetyValveOFF();
sets valve_relay to low.
Sets safety_valve_state to false.

void safetyValveON();
sets valve_relay to high.
Sets safety_valve_state to true.

void pumpOFF();
sets pump_relay to low.
Calls safetyValveOFF();
Sets pump_state to false.
Sets buttonA_ledRing to low.

void pumpON();
sets pump_relay to high.
Calls safetyValveON();
Sets pump_state to true.
Sets buttonA_ledRing to high.

debounce();
if the button trigger(trigger_X) is true, it sets the corresponding state to the opposite of what it was before.
The checkHardwareStateChange(); checks the state later and acts accordingly. (true == thing turns on, and the other way around).

We need to do this instead of calling the ON or OFF function directly, because the idlestate(); has to go to a complete stop, otherwise the ledrings on the button will shut down themselves right after they light up.

```
void debounce() {
    if (trigger_A) {

        if ((millis() - buttonAStartMillis) > debounceDelay) {
            //if pumpstate is true, set to false and vice versa.
            if (pump_state ? pump_state = false : pump_state = true);
        }
        buttonAStartMillis = millis();
        trigger_A = false;
    }
```

void valveControllerButton()
If the valve is true, it sets it to false and vice versa.
True == Open
False == Closed

darinValveOpen()
Heres a lazy special, instead of creating an object once and pass it around, it's created in both open and closed function. This mostly because of time-pressure and it had to be passed around between 10 different functions to work. So it's a technical debt I guess.

Servo is created. Servo writes 4 for closed. This is because at 0 the servo starts to twitch, but seems nice and closed at 4.

90 is open because of 90° degrees is the amount we want. Do not override the 90° since the physical construction can't handle more.

```
void drainValveOpen() {
    Servo servo;
    servo.attach(drain_valve_pwm_pin);
    servo.write(4);
    digitalWrite(buttonC_ledRing, HIGH);
    drain_valve_state = true;
//  Serial.println(pot_1);
}
//Closes the drain valve with a servo connected under the table
void drainValveClosed() {
    Servo servo;
    servo.attach(drain_valve_pwm_pin);
    servo.write(90);
```

```
    //analogWrite(drain_valve_pwm_pin, 0);
    digitalWrite(buttonC_ledRing, LOW);
    drain_valve_state = false;
   // Serial.println("{\"DRAIN_VALVE\":\"CLOSED\"}");
}
```

void readButtons()

If any button is triggered, corresponding button function is called. Switched to true in buttonIsTriggered function, and the debounce will handle the correct action.

```
//If any button is triggered, corresponding button activates.
if (digitalRead(button_A) == HIGH) {
    buttonAIsTriggered();
}
if (digitalRead(button_B) == HIGH) {

    buttonBIsTriggered();
}
if (digitalRead(button_C) == HIGH) {

    buttonCIsTriggered();
}
```

readPotentiometers()

Gets value from the potentiometers and maps it to new PWM value. Had to put constrains because the mapping was not always right for some reason...Tech debt I guess.

his is the servo value, it goes from five because at zero it acts weird, 5 seem to be a good start. 100 is straight up aka fully opened.

```
new_pot_3 = map(analogRead(potPin3), 150, 1000, 0, 90);
new_pot_3 = constrain(new_pot_3, 5, 100);
```

welcomeBlink()

The startup blink. blinks 3 times and then steps from left to right button led. its marvelous.

```
//there is a delay that is semi blocking but its for 5 milliseconds so it should be ok.
//but figure out a way to do it with millis would be optimal.
```

Classes:

SensorReadings:

.h file:

```
class Sensor{
private:
    int flowSensor;
    int lastFlowSensor;
public:
    //SETTER
    float setFlowSensorValue(float s);
    float setLastFlowSensorValue(float s);
    //GETTER
    float getFlowSensorValue() const;
};
```

.cpp:

setFlowensorValue(int s); sets corresponding float value to each object.
setLastFlowensorValue(int s); sets corresponding value from flowSensor.

getFlowensorValue(int s); gets corresponding value from each object.

```
float Sensor::setFlowSensorValue(float s){
    this -> flowSensor = s;
}

float Sensor::getFlowSensorValue() const {
    return this -> flowSensor;
}

float Sensor::setLastFlowSensorValue(float s) {
    this -> lastFlowSensor = s;
}
```

Convert:

The sensor:
The flow meter, for every liter of liquid passing through it per minute, it outputs about 4.5 pulses. This is due to the changing magnetic field caused by the magnet attached to the rotor shaft. Each pulse is approximately 2.25 milliliters.

.h file:

```
class Convert{
private:
    int X;
    int Y;
    float TIME = 0;
    float FREQUENCY = 0;
    float WATER = 0;
    float LS = 0;
    float TOTAL = 0;
public:
    float convertToLitres(float inflow);
```

.cpp file:

pulseIn meassures high and low.
Time is first set to equal all high-time and all low-time.

Litres = Q * time elapsed (seconds) / 60 (seconds/minute)
Litres = (Frequency (Pulses/second) / 7.5) * time elapsed (seconds) / 60
Litres = Pulses / (7.5 * 60)
Returns a float total that's litres per minute.

```
float Convert::convertToLitres(float inflow){
    this ->X = pulseIn(inflow, HIGH);
    this ->Y = pulseIn(inflow, LOW);
    this->TIME = X + Y;
    this->FREQUENCY = 1000000/this->TIME;
    this->WATER = this->FREQUENCY/7.5;
   this->LS = this->WATER/60;
   this->TOTAL = this->TOTAL + this->LS;
    return this-> TOTAL;
}
```

Formula:
Here we have determined flow rate by change in velocity of water. The pipe's cross-sectional area is known and remains constant, the average velocity is an indication of the flow rate. The basis relationship for determining the liquid's flow rate in such cases is **Q=VxA**, where **Q** is flow rate/total flow of water through the pipe, **V** is average velocity of the flow and **A** is the cross-sectional area of the pipe (viscosity, density and the friction of the liquid in contact with the pipe also influence the flow rate of water).

- Pulse frequency (Hz) = 7.5Q, Q is flow rate in Litres/minute
- Flow Rate (Litres/Min) = (Pulse frequency) / 7.5Q

- Sensor Frequency (Hz) = 7.5 * Q (Liters/min)
- Litres = Q * time elapsed (seconds) / 60 (seconds/minute)
- Litres = (Frequency (Pulses/second) / 7.5) * time elapsed (seconds) / 60
- Litres = Pulses / 7.5

*Error range +-10

Keep in mind that i however, am not a physicist. / Robin