Doser Mega2560

1.1 System Architecture

The system architecture consists of a microcontroller that manages sensor data acquisition through `GetSensorReading()` and calibrates sensors via `Calibration()`. It schedules tasks with `scheduleFun()` and handles time display on the Nextion interface using `setTimeNex()` and `runTimeNex()`. Sensor data is formatted as JSON using `sensorjson()`, while `AutoFun()` automates control processes based on sensor readings, and `nexVal()` processes user inputs from the display.

1.2 Libraries Used

- **DFRobot_PH**: A library for interfacing with pH sensors from DFRobot. It helps read and calibrate pH values from a pH probe.
- **DFRobot_EC**: Similar to the pH library, this is used to interface with electrical conductivity (EC) sensors, often used to measure the concentration of dissolved salts in water.
- **EEPROMex**: An extended version of the EEPROM library, allowing for larger memory spaces and better wear-leveling. It helps store settings like sensor calibration values and setpoints, ensuring they persist across power cycles.
- **ArduinoJson**: A powerful library used for parsing and creating JSON objects. It's useful for structuring data before sending it to external devices or storing it.
- **RTClib**: A library for interacting with real-time clock (RTC) modules, allowing your system to keep track of the date and time, even when powered off.
- **Time** and **TimeLib**: Both are libraries that help manage and manipulate time within your program, including time-stamping events or triggering tasks after set intervals.
- **MemoryFree**: A debugging library used to monitor how much dynamic memory (RAM) is free at any point in your program. This is essential when you're working with limited memory, like in microcontrollers.
- **SD**: The library used to read from and write to SD cards. It enables data logging, where you can store sensor readings, system status, or error logs over time.
- **Wire**: This library is the backbone for I2C communication, allowing the microcontroller to communicate with sensors and other devices over I2C.
- **menu**: Likely a library for handling and managing menu-driven interfaces, making it easier to interact with multiple settings and options in a user-friendly way (e.g., using an LCD screen).
- **TimerOne**: A library for controlling hardware timers on Arduino. This allows for precise timing control, such as scheduling a task to run at exact intervals.
- **TaskScheduler**: A flexible task scheduling library that helps run multiple tasks with precise timing, allowing for multitasking without blocking code.
- **DallasTemperature**: A library for interfacing with Dallas Semiconductor's DS18B20 temperature sensors, often used for precise water temperature measurements.
- **OneWire**: The communication protocol used by DallasTemperature sensors and other devices that operate over a single data line.
- **GravityTDS**: This is likely the Total Dissolved Solids (TDS) sensor library, used for measuring the concentration of dissolved substances in the water, such as salts, minerals, and metals.

1.3 Pin Definitions

- **BUZZER, LEDS, and Relays:**
  - `BUZZER (Pin 13)`: This pin controls a buzzer, likely to alert you to certain events.
  - `LEDPOWER, LEDRED, LEDGREEN, LEDBLUE (Pins 10, 11, 9, 12)`: These are likely part of an RGB LED indicator system for status feedback.
  - **Relays:**
    - `TPUMPR (Pin A15)`: Controls a pump.
    - `HEATERR (Pin A12)`: Controls a heater.
    - `MIXERR (Pin A13)`: Controls a mixer.
    - `RORELAYR (Pin A14)`: Activates the RO (Reverse Osmosis) water system.

Ultrasonic Sensor (Water Level)

- **Pins:**
  - `trigPin (Pin 46)`: Trigger pin of the ultrasonic sensor for water level measurement.
  - `echoPin (Pin 48)`: Echo pin of the ultrasonic sensor.

Sensors

- **pH and EC sensors:**
  - `PH_PIN (Pin A0)`: Reads the pH sensor.
  - `EC_PIN (Pin A1)`: Reads the EC sensor for electrical conductivity.
- **Gravity TDS (Total Dissolved Solids) Sensor**: Initialized as `gravityTds`.
- **Dallas Temperature Sensor:**
  - `ONE_WIRE_BUS (Pin 1)`: The 1-Wire bus used for temperature readings, typically for monitoring water temperature.

Stepper Motor Control for pH and EC Dosing

- **Pin Definitions:**
  - `PHLOW_STEP_PIN (Pin 3)`,`PHLOW_DIR_PIN (Pin 4)`: Stepper motor control for the low pH dosing.
  - `PHHIGH_STEP_PIN (Pin 23)`,`PHHIGH_DIR_PIN (Pin 22)`: Stepper motor control for the high pH dosing.
  - `EC_A_STEP_PIN (Pin 5)`,`EC_A_DIR_PIN (Pin 6)`: Stepper motor control for EC dosing pump A.
  - `EC_B_STEP_PIN (Pin 7)`,`EC_B_DIR_PIN (Pin 8)`: Stepper motor control for EC dosing pump B.
  - `EC_C_STEP_PIN (Pin 9)`,`EC_C_DIR_PIN (Pin 10)`: Stepper motor control for EC dosing pump C.

SD Card

- `SD_DETECT_PIN (Pin 53)` and `SDSS (Pin 53)`: Used to manage the SD card, which might be used for data logging (e.g., recording sensor readings or system events).

Doser Mega2560

EEPROM

The system uses the **EEPROMex** library, possibly for storing sensor calibration settings or user configuration values.

RTC (Real-Time Clock)

The `rtc` object uses the DS3231 RTC module to keep track of time, which is essential for time-stamping sensor data or automating scheduled tasks.

Constants

- `Offsetec` and `Offset`: These are compensation values for the EC and pH readings, respectively, allowing for deviation correction in sensor readings.
- **Sampling/Print Intervals**:
  - `samplingInterval (20 ms)`: Frequency for sensor readings.
  - `printInterval (800 ms)`: Frequency for printing the data (likely to Serial or SD card).
- `ArrayLenth (5)`: The number of samples collected before an average is calculated.

1.4 Functions

1. **`GetSensorReading()`**:

- Purpose: Reads sensor values such as pH, EC, temperature, TDS, etc., from the connected sensors (GravityTDS, pH, EC, DallasTemperature).
- Example actions: Collect data from the sensors and store or process it.

2. **`Calibration()`**:

- Purpose: Handles the calibration of the pH, EC, and possibly other sensors to ensure accurate measurements.
- Example actions: Implement calibration logic to adjust sensor readings based on reference values.

3. **`scheduleFun()`**:

- Purpose: Likely handles tasks that need to be scheduled, such as recurring sensor readings, motor control, or other time-based operations.
- Example actions: Utilize the `TaskScheduler` library to schedule tasks at specific intervals.

4. **`setTimeNex()`**:

- Purpose: Manages setting time in the system, likely involving the Nextion display for user input and adjusting the RTC module (DS3231).
- Example actions: Capture user input for time/date and configure the RTC.

5. **`runTimeNex()`**:

- Purpose: Displays the current time on the Nextion display, continuously updating it from the RTC.
- Example actions: Retrieve the current time from the RTC and show it on the Nextion display.

6. **`sensorjson()`**:

- Purpose: Collects sensor data, formats it into JSON, and perhaps publishes it (e.g., to an SD card, network, or cloud service).
- Example actions: Gather sensor readings, package them into a JSON object using the `ArduinoJson` library.

7. **`AutoFun()`**:

- Purpose: Controls the automatic functionality of the system based on sensor readings (e.g., adjusting pumps or relays based on pH, EC, or water level).
- Example actions: Implement automatic control logic that triggers pumps or dosing mechanisms based on predefined thresholds.

8. **`nexVal()`**:

- Purpose: Possibly reads and processes values from the Nextion display, such as user inputs or commands.
- Example actions: Capture user inputs via the Nextion display, like setpoints or mode changes, and update the system state accordingly.

1.5 Function Breakdown

1.5.1 void setup()

This function initializes the system's hardware and software components. It sets up communication, configures pins, and initializes sensors and actuators.

```
void setup()

{

  Serial.begin(115200);

  Serial3.begin(115200);

  Serial2.begin(9600); // Nextion display communication
```

Doser Mega2560

```
sensors.begin();    // Dallas Temperature Sensor

ph.begin();         // Initialize pH sensor
ec.begin();         // Initialize EC sensor

// Configure output pins for pH and EC stepper motors
pinMode(PHLOW_STEP_PIN, OUTPUT);
pinMode(PHLOW_DIR_PIN, OUTPUT);
pinMode(PHHIGH_STEP_PIN, OUTPUT);
pinMode(PHHIGH_DIR_PIN, OUTPUT);
pinMode(EC_A_STEP_PIN, OUTPUT);
pinMode(EC_B_STEP_PIN, OUTPUT);
pinMode(EC_C_STEP_PIN, OUTPUT);
pinMode(EC_A_DIR_PIN, OUTPUT);
pinMode(EC_B_DIR_PIN, OUTPUT);
pinMode(EC_C_DIR_PIN, OUTPUT);

// Default motor direction
digitalWrite(EC_A_DIR_PIN, LOW);
digitalWrite(EC_B_DIR_PIN, LOW);
digitalWrite(EC_C_DIR_PIN, LOW);

// Set up other hardware components
pinMode(BUZZER, OUTPUT);
pinMode(LEDRED, OUTPUT);
pinMode(LEDGREEN, OUTPUT);
pinMode(LEDBLUE, OUTPUT);
pinMode(LEDPOWER, OUTPUT);
digitalWrite(LEDPOWER, HIGH);
```

```
  pinMode(TPUMPR, OUTPUT);
  pinMode(HEATERR, OUTPUT);
  pinMode(MIXERR, OUTPUT);
  pinMode(RORELAYR, OUTPUT);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT_PULLUP);

  // Initialize sensor readings
  readalldata();

  // Dissolved oxygen probe initialization
  if (!probeDo) {
    getNutrientDO();
  }

  pinMode(resetPin, OUTPUT);  // Reset pin setup
}
```

1.5.2 **void sensorjson()**

This function collects sensor data, packages it into a JSON object, and sends it over serial interfaces for monitoring or external processing.

```
void sensorjson() {
  StaticJsonDocument<500> doc;

  // Add sensor values to JSON
  doc["pH"] = pH;
  doc["EC"] = ECValueavrg;
  doc["NT"] = nutrientTemperature;
  doc["DO"] = DOvalue;
  doc["NTankLevel"] = waterconlevel;
```

```
  // Add system state values
  doc["Sampler"] = sampumpSt;
  doc["Heater"] = HeaterSt;
  doc["Mixer"] = mixerSt;
  doc["RO"] = RoSt;

  // Add setpoint values
  doc["pHSetPA"] = pHAutoSP;
  doc["ECSetPA"] = ECAutoSP;

  // Serialize and send JSON data
  serializeJson(doc, Serial3);
  serializeJsonPretty(doc, Serial);
  Serial.println();
}
```

1.5.3 **void GetSensorReading()**

This function handles the process of reading sensor data for pH, EC, temperature, and other measurements. It also sends the sensor data as JSON.

```
void GetSensorReading() {
  sensorjson();
  delay(1000);  // Wait for 1 second
  getNutrientPHavg();  // Read average pH value
  delay(1000);  // Wait for 1 second
  getNutrientECavg();  // Read average EC value
  delay(1000);  // Wait for 1 second
  getNutrientTemperature();  // Read temperature
  delay(1000);  // Wait for 1 second
}
```

1.5.4 bool readSerial(char result[])

Reads incoming serial data from the Serial interface and stores it into a string. It looks for newline characters (\n) as the end of the input.

```
bool readSerial(char result[]) {
  while (Serial.available() > 0) {
    char inChar = Serial.read();
    if (inChar == '\n') {
      result[i] = '\0';  // End the string
      Serial.flush();    // Clear the buffer
      i = 0;             // Reset index
      return true;       // Successfully read data
    }
    if (inChar != '\r') {  // Ignore carriage return
      result[i] = inChar;  // Store character
      i++;
    }
    delay(1);  // Short delay for buffer management
  }
  return false;  // No data read
}
```

1.5.5 **void loop()**

This is the main program loop that repeatedly executes tasks at different intervals, such as reading sensors, updating the time, and handling events.

```
void loop()
{
  unsigned long currentMillis = millis();

  // Read sensors and update time every 10 seconds
  if (currentMillis - previousMillis1 >= 10000) {
```

```
    previousMillis1 = currentMillis;
    GetSensorReading();  // Collect sensor data
    setTimeNex();        // Update time on the Nextion display
    runTimeNex();        // Run Nextion display logic
    nexVal();            // Process Nextion display values
    SerialCom();         // Serial communication handling
  }

  // Manual readings every 3 seconds
  if (currentMillis - previousMillis2 >= 3000) {
    previousMillis2 = currentMillis;
    manualread();  // Read manual inputs
  }

  // Event mode every 4 seconds
  if (currentMillis - previousMillis3 >= 4000) {
    previousMillis3 = currentMillis;
    eventmode();  // Handle event modes
  }

  // Calibration every 45 seconds
  if (currentMillis - previousMillis4 >= 45000) {
    previousMillis4 = currentMillis;
    Calibration();  // Perform sensor calibration
  }
}
```

1.5.6 **roundDecimalPoint**

This function rounds a floating-point number to a specified number of decimal places.

```
float roundDecimalPoint(float in_value, int decimal_place) {
  float multiplier = powf(10.0f, decimal_place);
  in_value = roundf(in_value * multiplier) / multiplier;
  return in_value;
}
```

1.5.7 **avergearrayec**

This function calculates the average of an integer array, excluding the highest and lowest values, which is useful for error-tolerant averaging.

```
double avergearrayec(int * arr, int number) {
  int i, max, min;
  double avg;
  long amount = 0;
  if (number <= 0) {
    Serial.println("Error number for the array to averaging!\n");
    return 0;
  }
  if (number < 5) {
    for (i = 0; i < number; i++) {
      amount += arr[i];
    }
    avg = amount / number;
    return avg;
  } else {
    if (arr[0] < arr[1]) {
      min = arr[0];
```

```
    max = arr[1];
   } else {
    min = arr[1];
    max = arr[0];
   }
   for (i = 2; i < number; i++) {
    if (arr[i] < min) {
     amount += min;
     min = arr[i];
    } else if (arr[i] > max) {
     amount += max;
     max = arr[i];
    } else {
     amount += arr[i];
    }
   }
   avg = (double)amount / (number - 2);
  }
  return avg;
}
```

1.5.8 **avergearray**

This function is identical to avergearrayec, except it’s a general-purpose average function for integer arrays.

```
double avergearray(int * arr, int number) {
```

```
  // Similar implementation to avergearrayec

}
```

1.5.9 **void timeSec()**

These functions parse specific time components (seconds, minutes, hours, date, month, year) from a string received from the Nextion display.

```
void timeSec() {

  val = dfd.indexOf("s") + 1;

  dfd.remove(0, val);

  uint8_t secRc = dfd.toInt();

  secsrc = secRc;

}
```

1.5.10 **timeMin()**

Extracts minutes from the buff string.

```
int timeMin() {

  char *cmn = strtok(buff, "s");

  String cSmn = cmn;

  val = cSmn.indexOf("n") + 1;

  cSmn.remove(0, val);

  uint8_t minRc = cSmn.toInt();

  minrc = minRc;

  return (minRc);

}
```

1.5.11 void timeHr()

Extracts hours from the buff string.

```
void timeHr() {
  char *chr = strtok(buff, "n");
  String cShr = chr;
  val = cShr.indexOf("h") + 1;
  cShr.remove(0, val);
  uint8_t hrRc = cShr.toInt();
  hourrc = hrRc;
}
```

1.5.12 **void** timeDt()

Extracts the date (day) from the buff string.

```
void timeDt() {
  char *cdt = strtok(buff, "m");
  String cSdt = cdt;
  val = cSdt.indexOf("d") + 1;
  cSdt.remove(0, val);
  uint8_t dtRc = cSdt.toInt();
  Daterc = dtRc;
}
```

1.5.13 **void timeMnt()**

Extracts the month from the buff string.

```
void timeMnt() {
  char *cmnt = strtok(buff, "y");
  String cSmnt = cmnt;
  val = cSmnt.indexOf("m") + 1;
  cSmnt.remove(0, val);
  uint8_t mntRc = cSmnt.toInt();
```

```
  Monthrc = mntRc;
}
```

1.5.14 **void timeYr()**

Extracts the year from the buff string.

```
void timeYr() {
  char *cyr = strtok(buff, "h");
  String cSyr = cyr;
  val = cSyr.indexOf("y") + 1;
  cSyr.remove(0, val);
  int yrRc = cSyr.toInt();
  Yearrc = yrRc;
}
```

1.5.15 **void dateandtime()**

This function retrieves the current time and date from an RTC module.

```
void dateandtime() {
  DateTime now = rtc.now();
  hourupg = now.hour();
  minupg = now.minute();
  secslive = now.second();
  Datelive = now.day();
  Monthlive = now.month();
  Yearlive = now.year();
}
```

1.5.16 **void DoserAuto()**

This function automates pH and EC dosing based on setpoints, using hysteresis for control.

Doser Mega2560

```
void DoserAuto() {
  static unsigned long samplingTime = millis();
  static unsigned long printTime = millis();
  Serial.println("DoserAuto() is running.");

  // EEPROM read functions
  EepromReadPHCal();
  EepromReadEC();

  // Turn on Green LED indicating the function is running
  digitalWrite(LEDGREEN, HIGH);

  // Display setpoints and hysteresis values
  Serial.println(pHHys);
  Serial.print("pH SETPOINT: ");
  Serial.println(pHAutoSP);
  Serial.print("EC SETPOINT: ");
  Serial.println(ECAutoSP);

  // Calculate pH and EC
  pH = roundDecimalPoint(phValueavrg, 2);
  EC = roundDecimalPoint(ecValue, 2);

  // Compare with setpoints and adjust dosing systems accordingly
  // Check pH, EC ranges and trigger appropriate dosing motors

  // Example for pH adjustment
  if (pH < HysterisMin) {
    digitalWrite(PHHIGH_STEP_PIN, HIGH);  // Increase pH
  } else if (pH > HysterisPlus) {
```

```
    digitalWrite(PHLOW_STEP_PIN, HIGH);  // Decrease pH
  }

  // Similar logic is applied for EC adjustment
  ...
}
```

1.5.17 **manualdosemin()**

This function controls the dosage of a solution to decrease the pH level. It uses a stepper motor to deliver a specified amount of the solution based on the phplus variable.

```
result manualdosemin() {
  digitalWrite(LEDBLUE, HIGH);
  int RPHUP = PHUPR * 230; // Calculate steps for the motor
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis > pinTime) {
    previousMillis = currentMillis;
    digitalWrite(PHHIGH_DIR_PIN, LOW); // Set direction
    for (int y = 0; y < phplus; y++) { // Repeat for the amount to dose
      for (int x = 0; x < RPHUP; x++) {
        digitalWrite(PHHIGH_STEP_PIN, HIGH);
        delayMicroseconds(Speed); // Control speed
        digitalWrite(PHHIGH_STEP_PIN, LOW);
        delayMicroseconds(Speed);
        Serial.println("PHHIGH_STEP_Speed: ");
        Serial.println(Speed);
      }
```

```
    }
    pinTime = pinLowTime; // Reset timer
  }
}
```

1.5.18 **manualdoseplus()**

This function controls the dosage of a solution to increase the pH level, utilizing a stepper motor similar to manualdosemin().

```
result manualdoseplus() {
  digitalWrite(LEDBLUE, HIGH);
  int RPHDOWN = PHDOWNR * 230; // Calculate steps for the motor
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis > pinTime) {
    previousMillis = currentMillis;
    digitalWrite(PHLOW_DIR_PIN, LOW); // Set direction
    for (int y = 0; y < phmin; y++) { // Repeat for the amount to dose
      for (int x = 0; x < RPHDOWN; x++) {
        digitalWrite(PHLOW_STEP_PIN, HIGH);
        delayMicroseconds(Speed); // Control speed
        digitalWrite(PHLOW_STEP_PIN, LOW);
        delayMicroseconds(Speed);
        Serial.println("PHLOW_STEP_Speed: ");
        Serial.println(Speed);
      }
    }
    pinTime = pinLowTime; // Reset timer
  }
}
```

1.5.19 **manualdoseEcA()**

This function controls the dosing for Electrical Conductivity (EC) solution A using a stepper motor.

```
int ECRA; // Global variable for EC A ratio
result manualdoseEcA() {
  Serial.println("ecaaa");
  digitalWrite(LEDBLUE, HIGH);
  int ECRA = ECRatioA * 230; // Calculate steps for the motor
  digitalWrite(EC_A_DIR_PIN, LOW); // Set direction
  for (int z = 0; z < ECA; z++) { // Repeat for the amount to dose
    for (int x = 0; x < ECRA; x++) {
      digitalWrite(EC_A_STEP_PIN, HIGH);
      delayMicroseconds(Speed); // Control speed
      digitalWrite(EC_A_STEP_PIN, LOW);
      delayMicroseconds(Speed);
      Serial.println("EC_A_STEP_Speed: ");
      Serial.println(Speed);
    }
  }
}
```

1.5.20 **manualdoseEcB()**

This function manages the dosing for Electrical Conductivity (EC) solution B using a stepper motor.

```
int ECRB; // Global variable for EC B ratio
result manualdoseEcB() {
  digitalWrite(LEDBLUE, HIGH);
  int ECRB = ECRatioB * 240; // Calculate steps for the motor
  digitalWrite(EC_B_DIR_PIN, LOW); // Set direction
  digitalWrite(EC_C_DIR_PIN, LOW); // Set direction for EC C
  for (int b = 0; b < ECB; b++) { // Repeat for the amount to dose
```

```
    for (int x = 0; x < ECRB; x++) {
      digitalWrite(EC_B_STEP_PIN, HIGH);
      delayMicroseconds(Speed); // Control speed
      digitalWrite(EC_B_STEP_PIN, LOW);
      delayMicroseconds(Speed);
      Serial.println("EC_B_STEP_Speed: ");
      Serial.println(Speed);
    }
  }
}
```

1.5.21 void SheduleDoser(float pHAutoSetValue, float pHHys, float ECAutoSetValue, float ECHys)

This function schedules the dosing process for both pH and EC based on input parameters and the current values of pH and EC.

```
void SheduleDoser(float pHAutoSetValue, float pHHys, float ECAutoSetValue, float ECHys ) {

  digitalWrite(LEDGREEN, HIGH); // Indicate dosing is active

  digitalWrite(TPUMPR, LOW); // Activate pump

  digitalWrite(MIXERR, LOW); // Activate mixer



  pHSetpoint = pHAutoSetValue;

  phSetHysteris = pHHys;

  pH = roundDecimalPoint(phValue, 2); // Read current pH



  ECSetpoint = ECAutoSetValue;

  ECHys = ECSetHysteris;

  EC = roundDecimalPoint(ecValue, 2); // Read current EC
```

```
  // Calculate hysteresis limits
  float HysterisMin = (pHSetpoint - phSetHysteris);
  float HysterisPlus = (pHSetpoint + phSetHysteris);
  ECHysterisMin = (ECSetpoint - ECSetHysteris);
  ECHysterisPlus = (ECSetpoint + ECSetHysteris);


  // Dosing logic for pH and EC
  if (StopPHHys == false) {
    // Logic for adjusting pH
    // ... (additional dosing conditions and actions)
  }
}
```

1.5.22 **void Calibration()**

This function reads voltage from pH and EC sensors every second and prints the results. It also listens for serial commands to calibrate the sensors.

```
/************************************ calibration **********************************/
void Calibration() {
  char cmd[10];
  static unsigned long timepoint = millis();


  // Calibration process executed every second
  if (millis() - timepoint > 1000U) {  // time interval: 1s
    timepoint = millis();
```

```
    // Read the pH voltage from the sensor

    voltagePH = analogRead(PH_PIN) / 1024.0 * 5000;  // read the pH voltage

    phValue = ph.readPH(voltagePH, temperature);  // convert voltage to pH with temperature
compensation

    Serial.print("pH:");

    Serial.print(phValue, 2);



    // Read the EC voltage from the sensor

    voltageEC = analogRead(EC_PIN) / 1024.0 * 5000;

    ecValue = ec.readEC(voltageEC, temperature);  // convert voltage to EC with temperature
compensation

    Serial.print(", EC:");

    Serial.print(ecValue, 2);

    Serial.println("ms/cm");

    Serial.println();

  }



  // Check for calibration commands via serial input

  if (readSerial(cmd)) {

    toUpperCase(cmd);

    if (strstr(cmd, "PH")) {

      Serial.println("Entering pH calibration");

      ph.calibration(voltagePH, temperature, cmd);  // pH calibration process by Serial CMD

    }

    if (strstr(cmd, "EC")) {
```

```
    ec.calibration(voltageEC, temperature, cmd);  // EC calibration process by Serial CMD
   }
  }
}
```

1.5.23 **getNutrientTDS()**

This function computes the TDS from the EC value and rounds it to two decimal places.

```
float getNutrientCF() {
  float cfValue = ecValue * 10;  // Calculate conductivity factor (CF) based on EC value
  nutrientCF = cfValue;
  return cfValue;
}
```

1.5.24 **getNutrientCF()**

This function computes the CF from the EC value and stores it for further use.

```
float getNutrientCF() {
  float cfValue = ecValue * 10;  // Calculate conductivity factor (CF) based on EC value
  nutrientCF = cfValue;
  return cfValue;
}
```

1.5.25 **getNutrientDO()**

This function reads the raw ADC value for DO, converts it to voltage, and retrieves the corresponding DO value.

```
float getNutrientDO() {
  Temperaturet = (uint8_t)READ_TEMP;  // Read temperature for DO calculation
```

```
  ADC_Raw = analogRead(DO_PIN);  // Read raw ADC value for dissolved oxygen
  ADC_Voltage = uint32_t(VREF) * ADC_Raw / ADC_RES;  // Convert raw value to voltage
  float DO = (readDO(ADC_Voltage, Temperaturet));  // Read dissolved oxygen value
  float DOvalue = DO;
  return DOvalue;
}
```

1.5.26 **void getNutrientTL()**

This function uses an ultrasonic sensor to measure the water level and controls the pump based on the distance measured. It manages pump states to prevent overflow or dry running.

```
void getNutrientTL() {
  digitalWrite(trigPin, LOW);  // Set trigger pin low
  delayMicroseconds(5);
  digitalWrite(trigPin, HIGH);  // Send a trigger pulse
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  duration = pulseIn(echoPin, HIGH);  // Read the echo pulse duration
  distanceincm = duration * 0.034 / 2;  // Calculate distance in cm
  distanceinInch = distanceincm / 2.54;  // Convert cm to inches
  volume = pi * 16 * distanceincm;  // Calculate volume based on distance
  waterHeight = tankHeight - distanceinInch;  // Calculate water height

  // Determine pump state based on water level
  int Wtemp;
  int tankSetHysteris = 10;  // Set tank hysteresis in inches
  int TankHysteris = (tanksetpoint - tankSetHysteris);
  if (distanceinInch < tanksetpoint && Wtemp == 0) {
```

```
    digitalWrite(RORELAYR, HIGH);  // Turn off pump
    Serial.println("Water PUMP OFF");
    Tankfull == false;
    Wtemp = 1;
  } else if (distanceinInch < tanksetpoint && Wtemp == 1) {
    digitalWrite(RORELAYR, HIGH);
    Serial.println("Water PUMP OFF");
    Tankfull == false;
  } else if (distanceinInch < TankHysteris) {
    digitalWrite(RORELAYR, LOW);  // Turn on pump
    Serial.println("Water PUMP ON");
    Tankfull == false;
    Wtemp = 0;
  } else if (distanceinInch < tankHeight) {
    Serial.println("Water PUMP ON");
    Tankfull == true;
    Wtemp = 0;
  }
  return distance;
}
```

1.5.27 **void getNutrientTL()**

This function uses an ultrasonic sensor to measure the water level and controls the pump based on the distance measured. It manages pump states to prevent overflow or dry running.

```
void getNutrientTL() {

  digitalWrite(trigPin, LOW);  // Set trigger pin low

  delayMicroseconds(5);

  digitalWrite(trigPin, HIGH);  // Send a trigger pulse

  delayMicroseconds(10);
```

```
digitalWrite(trigPin, LOW);


duration = pulseIn(echoPin, HIGH);  // Read the echo pulse duration
distanceincm = duration * 0.034 / 2;  // Calculate distance in cm
distanceinInch = distanceincm / 2.54;  // Convert cm to inches
volume = pi * 16 * distanceincm;  // Calculate volume based on distance
waterHeight = tankHeight - distanceinInch;  // Calculate water height


// Determine pump state based on water level
int Wtemp;
int tankSetHysteris = 10;  // Set tank hysteresis in inches
int TankHysteris = (tanksetpoint - tankSetHysteris);
if (distanceinInch < tanksetpoint && Wtemp == 0) {
  digitalWrite(RORELAYR, HIGH);  // Turn off pump
  Serial.println("Water PUMP OFF");
  Tankfull == false;
  Wtemp = 1;
} else if (distanceinInch < tanksetpoint && Wtemp == 1) {
  digitalWrite(RORELAYR, HIGH);
  Serial.println("Water PUMP OFF");
  Tankfull == false;
} else if (distanceinInch < TankHysteris) {
  digitalWrite(RORELAYR, LOW);  // Turn on pump
  Serial.println("Water PUMP ON");
```

```
    Tankfull == false;

    Wtemp = 0;

  } else if (distanceinInch < tankHeight) {

    Serial.println("Water PUMP ON");

    Tankfull == true;

    Wtemp = 0;

  }

  return distance;

}
```

1.5.28 **getNutrientTemperature()**

This function requests temperature readings from a Dallas temperature sensor and converts the value to Fahrenheit.

```
float getNutrientTemperature() {
  // Read DALLAS temperature sensor
  sensors.requestTemperatures();  // Send the command to get temperatures
  float ntValue = (sensors.getTempCByIndex(0));  // Get temperature from the first sensor
  ntValue = roundDecimalPoint(ntValue, 1);  // Round to one decimal point
  Fahrenheit = sensors.toFahrenheit(ntValue);  // Convert to Fahrenheit
  nutrientTemperature = ntValue;  // Store the nutrient temperature
  return ntValue;  // Return temperature in Celsius
}
```

1.5.29 **getNutrientPHavg()**

This function samples the pH sensor at specified intervals, calculates the average value, and checks for stability. It also handles calibration based on the average voltage.

```
float getNutrientPHavg() {
```

Doser Mega2560

```
static unsigned long samplingTime = millis();

static float voltagePHavrg, phtest;


// Sample pH value at specified intervals

if (millis() - samplingTime > samplingInterval) {

  pHArray[pHArrayIndex++] = analogRead(PH_PIN);  // Read pH value and store it

  if (pHArrayIndex == ArrayLenth)

    pHArrayIndex = 0;  // Reset index if it exceeds array length


  voltagePH = analogRead(PH_PIN) / 1024.0 * 5000;  // Read pH voltage

  phValue = (ph.readPH(voltagePH, nutrientTemperature)) + Offset;  // Read pH value with offset


  voltagePHavrg = avergearray(pHArray, ArrayLenth) / 1024.0 * 5000;  // Calculate average voltage

  phValueavrg = (ph.readPH(voltagePHavrg, nutrientTemperature)) + Offset;  // Average pH value


  phtest = roundDecimalPoint(phValue, 2);  // Round pH value

  int phRv = phtest;

  int phValueavrgO = roundDecimalPoint(phValueavrg, 2);


  samplingTime = millis();  // Update sampling time


  // Check if pH values are stable

  if (phRv == phValueavrgO) {

    Serial.println("PH Stable");
```

```
    pHAvg = true;
  } else {
    Serial.println("PH Not Stable");
    pHAvg = false;
  }
 }

 ph.calibration(voltagePHavrg, nutrientTemperature);  // Calibrate pH sensor
 pH = roundDecimalPoint(phValueavrg, 2);  // Store average pH value
 Serial.print("PH read SENSOR : ");
 Serial.println(pH);

 return pH;  // Return average pH value
}
```