UNIVERSIDADE ESTADUAL DE MATO GROSSO DO SUL

UEMS – Unidade Universitária de Nova Andradina

ACCO, Domini da Silva

MAEKAWA, Murilo Yukio de Oliveira

Sistema de Irrigação Automatizado baseado em Arduino para Horta Escolar Rural

Nova Andradina – MS

2025

ACCO, Domini da Silva

MAEKAWA, Murilo Yukio de Oliveira

Sistema de Irrigação Automatizado baseado em Arduino para Horta Escolar Rural

Relatório técnico apresentado à Universidade Estadual de Mato Grosso do Sul (UEMS), Unidade Universitária de Nova Andradina, como parte das atividades do projeto de extensão e desenvolvimento tecnológico voltado à automação da irrigação de hortas escolares.

Orientadoras: Prof.ª Dr Simone de F. Tonhão
Prof.ª Dr Amanda Cristina Davi Resende
Prof.ª Dr Ana Carolina de S. Ribas dos Reis
Prof.ª Dr Kátia Guerchi Gonzales

Nova Andradina – MS

2025

Conteúdo

1 Especificação do Hardware do Sistema

Esta seção apresenta os componentes utilizados no desenvolvimento do sistema de automação da irrigação, bem como a descrição técnica das conexões empregadas. O objetivo é fornecer uma visão completa da arquitetura de hardware, abordando os dispositivos, seus modos de comunicação e orientações para montagem do circuito.

1.1 Componentes Utilizados

A seguir, apresentam-se os principais dispositivos que compõem o sistema:

1.1.1 Microcontrolador (Arduino/Mega)

O microcontrolador é responsável por toda a lógica do sistema, realizando leituras dos sensores, gravação de dados, comunicação com módulos externos e acionamento dos atuadores. É o núcleo do projeto, coordenando a interação entre os componentes conectados por barramentos como I2C, SPI ou Serial.

1.1.2 Sensor de Umidade do Solo

Utilizado para medir a quantidade de água presente no solo. Pode operar por resistência elétrica ou pela variação capacitiva. Envia ao microcontrolador um valor analógico proporcional ao grau de umidade do solo.

1.1.3 Módulo RTC (Relógio de Tempo Real)

O módulo RTC mantém o registro correto de data e hora, mesmo na ausência de alimentação. Sua comunicação ocorre via barramento I2C, utilizando os pinos SDA e SCL.

1.1.4 Módulo SD Card

Permite o armazenamento de dados e registros gerados pelo sistema. Opera utilizando o protocolo SPI, sendo responsável pelo armazenamento dos arquivos de log e informações relevantes para análise posterior.

1.1.5 Display LCD 16x2 com Interface I2C

O display é utilizado para apresentar informações ao usuário, como umidade atual, hora e status de irrigação. A interface I2C reduz a quantidade de cabos necessários e facilita a montagem.

1.1.6 Módulo Relé

Responsável por acionar a bomba de água, funcionando como uma chave controlada eletronicamente. Isola o microcontrolador de cargas de maior tensão e corrente, garantindo segurança e integridade do sistema.

1.1.7 Válvula Solenoide

Executa a irrigação física do sistema, sendo ligada e desligada por meio do módulo relé. Sua alimentação elétrica é independente, utilizando fonte própria conforme sua especificação, no caso está sendo utilizado, tanto no vivieiro quanto na horta válvulas 3/4 com voltagem 12v.

1.2 Conexões e Arquitetura do Sistema

A montagem do hardware segue uma estrutura baseada nos protocolos de comunicação padronizados de cada módulo. A seguir, detalham-se as principais ligações utilizadas.

1.2.1 Barramento I2C

O barramento I2C (Inter-Integrated Circuit) é um protocolo de comunicação serial amplamente utilizado em sistemas embarcados por permitir a conexão de múltiplos dispositivos utilizando apenas dois fios: **SDA** (dados) e **SCL** (clock). No projeto, tanto o módulo RTC quanto o display LCD utilizam essa interface, compartilhando a mesma linha de comunicação sem a necessidade de portas exclusivas para cada periférico.

A Figura 1 ilustra um exemplo de ligação do display LCD ao Arduino Mega por meio do barramento I2C.

Exemplo de conexões no Arduino Mega:

- **SDA** $\rightarrow$ pino 20 (fio verde)
- **SCL** $\rightarrow$ pino 21 (fio azul)
- **VCC** $\rightarrow$ 5V (fio vermelho)
- **GND** $\rightarrow$ GND (fio preto)

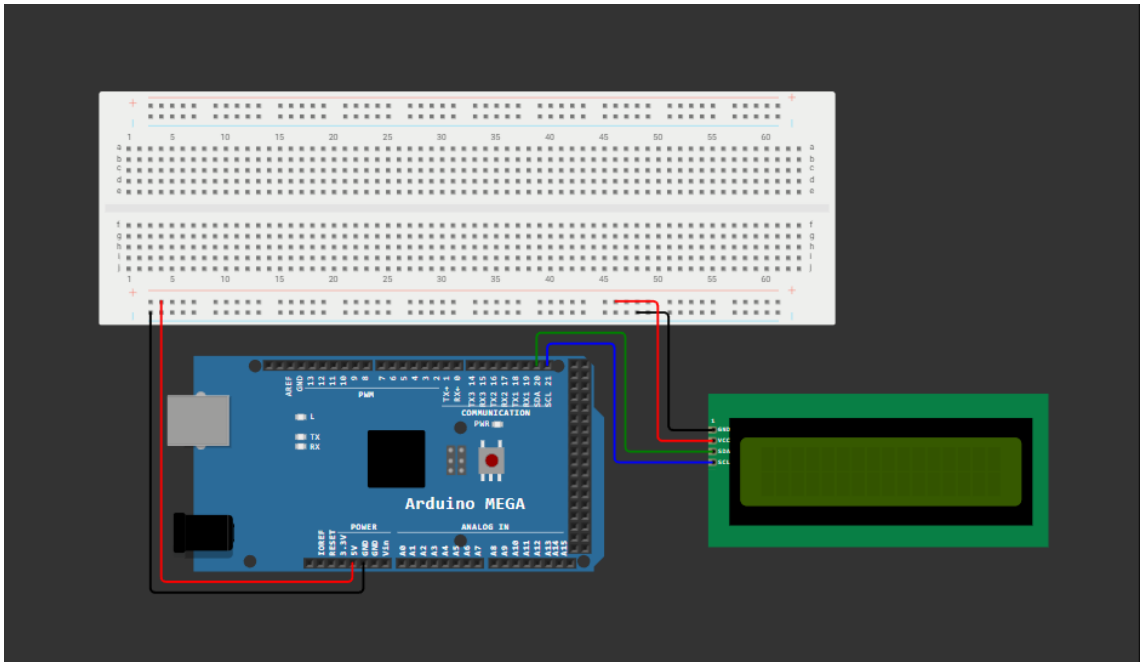

Figura 1: Exemplo de conexão do display LCD via I2C no Arduino Mega

1.2.2 Conexão do Módulo SD Card

O módulo SD Card utiliza o barramento SPI (Serial Peripheral Interface), um protocolo de comunicação síncrono muito empregado em sistemas embarcados devido à sua alta velocidade e simplicidade. Diferentemente do I2C, o SPI utiliza quatro linhas principais: **MISO**, **MOSI**, **SCK** e **CS**. Cada sinal possui uma função distinta no fluxo de dados entre o microcontrolador e o dispositivo externo.

No projeto, o módulo SD Card é responsável por registrar dados no cartão de memória, como leituras de sensores e eventos de irrigação. Para isso, ele é conectado diretamente ao barramento SPI do Arduino Mega, utilizando pinos dedicados para esse protocolo.

A Figura 2 apresenta um exemplo de ligação do módulo SD Card ao Arduino Mega por meio do barramento SPI.

Exemplo de conexões no Arduino Mega:

- **VCC** → 5V (fio vermelho)
- **GND** → GND (fio preto)
- **MISO (DO)** → pino 50 (fio amarelo)
- **MOSI (DI)** → pino 51 (fio azul)
- **SCK** → pino 52 (fio verde)
- **CS** → pino 4 (fio rosa)

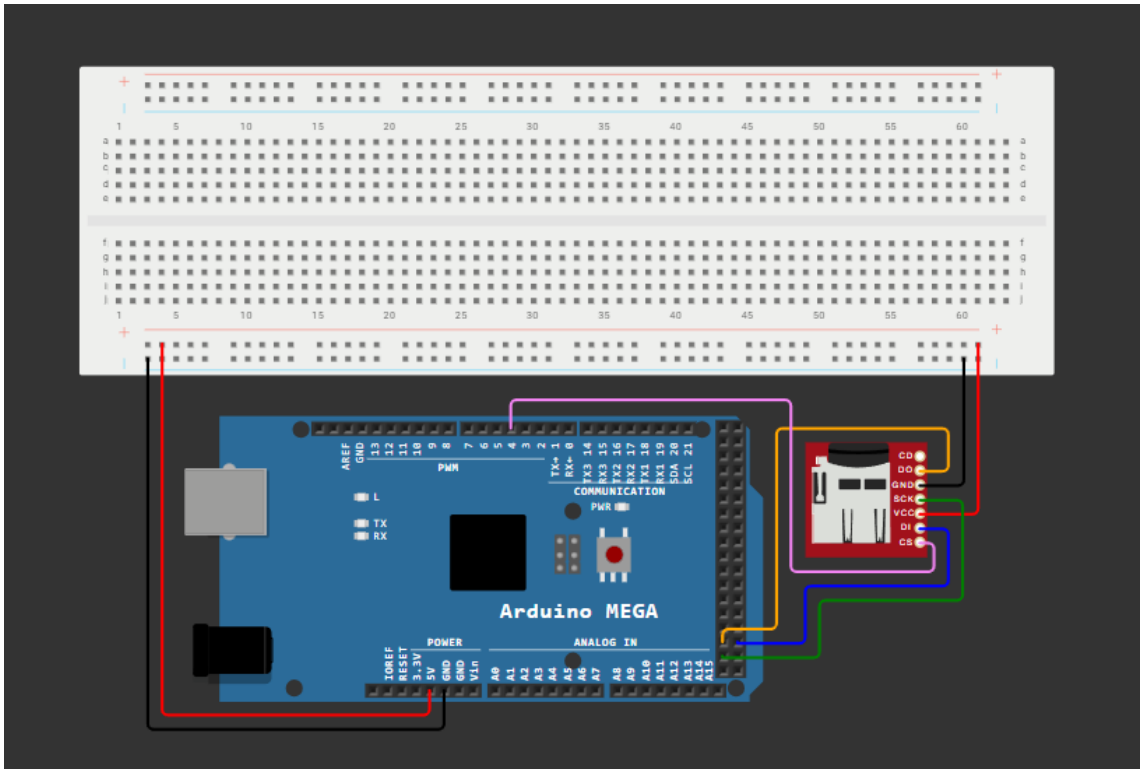

Figura 2: Exemplo de conexão do módulo SD Card ao Arduino Mega utilizando o barramento SPI

1.2.3 Sensor de Umidade do Solo

O sensor de umidade do solo é um dos principais componentes do sistema de automação da irrigação, responsável por medir o nível de umidade presente na terra e fornecer ao microcontrolador um valor analógico proporcional. Existem dois tipos principais de sensores: resistivos e capacitivos. Os sensores resistivos medem a variação da resistência elétrica do solo, porém apresentam menor durabilidade devido ao processo de corrosão dos eletrodos. Já os sensores capacitivos funcionam por variação de capacitância, oferecendo maior vida útil e maior estabilidade de leitura, sendo, portanto, a opção recomendada para sistemas de longo prazo.

Ligações típicas do sensor de umidade capacitivo:

- **Signal** $\rightarrow$ **A0** (entrada analógica utilizada para leitura do nível de umidade)
- **VCC** $\rightarrow$ **3.3V ou 5V** (dependendo do modelo específico do sensor)
- **GND** $\rightarrow$ **GND** (terra comum ao sistema)

Durante a operação, o sensor envia ao Arduino um sinal analógico que varia conforme a umidade do solo: valores mais altos indicam solo mais úmido, enquanto valores menores correspondem a solo mais seco. Esses valores são então utilizados pelo algoritmo de

controle para decidir quando acionar ou desligar a válvula solenoide, garantindo irrigação automática e eficiente.

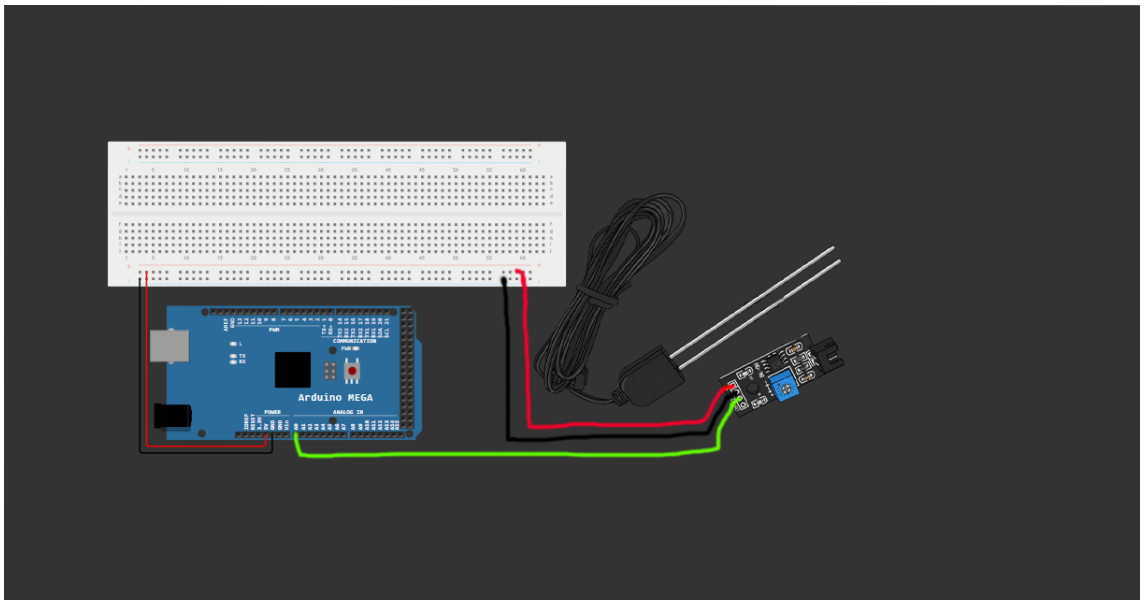

Figura 3: Exemplo de conexão do sensor de umidade ao Arduino Mega.

1.2.4 Módulo Relé e Acionamento da Válvula Solenoide

O módulo relé é o componente responsável por realizar a interface elétrica entre o microcontrolador e a válvula solenoide, permitindo que o Arduino acione dispositivos que operam com tensões e correntes superiores às suportadas por seus pinos digitais. Dessa forma, o relé funciona como um interruptor eletromecânico controlado pelo sinal enviado pelo Arduino.

Ligações típicas do módulo relé com o Arduino:

- **IN** $\rightarrow$ **D7** (pino digital utilizado para enviar o sinal de acionamento)
- **VCC** $\rightarrow$ **5V** (alimentação do módulo relé)
- **GND** $\rightarrow$ **GND** (referência de terra comum)

A válvula solenoide deve ser conectada aos terminais de potência do relé, normalmente identificados como **NO (Normally Open)** e **COM (Common)**. Quando o Arduino envia um sinal ao módulo relé, o contato entre NO e COM é fechado, permitindo a passagem da corrente para a válvula.

Como a válvula utiliza alimentação externa, é fundamental que:

- A fonte externa seja compatível com a tensão e corrente exigidas pela válvula.

- O terminal positivo da fonte seja conectado ao terminal **COM** do relé.
- O terminal **NO** seja ligado ao positivo da válvula.
- O fio negativo da válvula seja conectado ao negativo da fonte.
- Os terminais **GND** da fonte externa e do Arduino sejam interligados, garantindo uma *referência comum* para o acionamento correto.

A utilização do relé proporciona segurança ao sistema, evitando que a corrente elevada da válvula solenoide circule diretamente pelos pinos do Arduino, prevenindo danos ao microcontrolador.

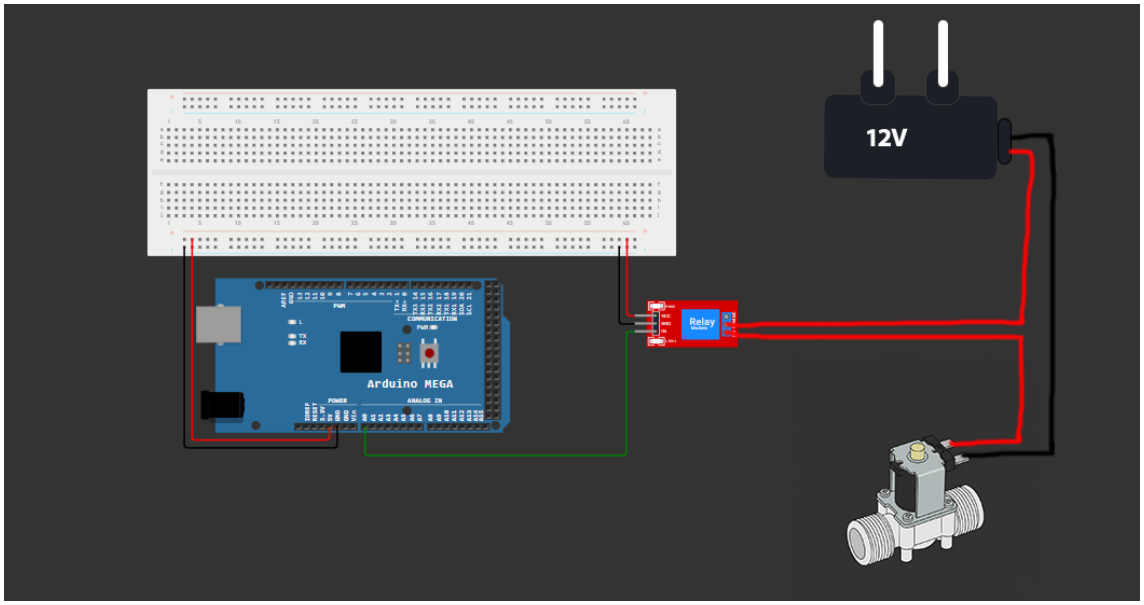

Figura 4: Exemplo de conexão da valvula solénoide ao Arduino Mega utilizando o relé.

1.3 Fonte de Alimentação

O microcontrolador e os módulos periféricos operam em 5V ou 3.3V. A bomba de água utiliza uma fonte dedicada, evitando sobrecarga no circuito principal. Medidas adicionais podem ser adotadas:

- Diodo de proteção (flyback).
- Fusível para proteção da linha.
- Capacitores para estabilização.

1.4 Esquema Geral de Funcionamento

O sistema opera de acordo com as seguintes etapas:

1. O sensor de umidade realiza a leitura do solo.
2. O módulo RTC fornece data e hora para registros.
3. O microcontrolador avalia as condições e decide se a irrigação será realizada.
4. Caso seja necessário, o relé é acionado, ligando a valvula solénoide.
5. Informações são exibidas no display LCD.
6. Dados são gravados no módulo SD Card.

2 Declarações e Configurações Iniciais

O trecho de código apresentado a seguir contém as bibliotecas e variáveis de configuração utilizadas no sistema Arduino. Essas definições são fundamentais para o correto funcionamento dos sensores, atuadores e módulos de comunicação, armazenamento e controle do sistema de automação.

```
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <EEPROM.h>
#include <RTClib.h>
#include <LiquidCrystal_I2C.h>
#include <avr/wdt.h>

RTC_DS3231 rtc;

const int chipSelect    = 4;
const int pinoSensor1   = A0;
const int pinoSensor2   = A1;
const int pinoValvula   = 9;

const int pinoSensorH1  = A2;
const int pinoSensorH2  = A3;
const int pinoSensorH3  = A4;
const int pinoValvulaH  = 10;

const int pinoFan       = 8;

```

```
23 const float R_DIV_TOP = 100000.0;
24 const float R_DIV_BOT = 33000.0;
25 const int pinBatSense = A6;
26 const int pinRelayPower = 7;
27 const bool RELAY_ACTIVE_HIGH = true;
28 const float BAT_LOW = 11.5;
29 const float BAT_RESTORE = 12.0;
30 unsigned long lastSwitchMillis = 0;
31 const unsigned long MIN_SWITCH_INTERVAL = 30000UL;
32
33 const int VALVE_OPEN_LEVEL   = LOW;
34 const int VALVE_CLOSED_LEVEL = HIGH;
35 const int FAN_ON_LEVEL  = HIGH;
36 const int FAN_OFF_LEVEL = LOW;
```

1: Configurações iniciais do sistema Arduino

2.1 Inclusão de Bibliotecas

O código inicia com a inclusão das principais bibliotecas necessárias para o funcionamento do sistema:

- **Wire.h**: possibilita a comunicação via protocolo I2C, utilizado para conectar sensores e periféricos como displays e relógios de tempo real.
- **SPI.h**: habilita a comunicação SPI (Serial Peripheral Interface), empregada na leitura e gravação em dispositivos como o cartão SD.
- **SD.h**: biblioteca responsável pelo controle do módulo de cartão SD.
- **EEPROM.h**: permite armazenar dados de forma permanente na memória EEPROM do microcontrolador.
- **RTClib.h**: utilizada para operar o módulo de relógio de tempo real (RTC), neste caso o modelo DS3231.
- **LiquidCrystal_I2C.h**: permite o uso simplificado de displays LCD por meio da interface I2C, economizando pinos de conexão.
- **avr/wdt.h**: fornece acesso ao *watchdog timer*, um mecanismo de segurança que reinicia o microcontrolador caso ocorra falha no código.

2.2 Configuração do Relógio de Tempo Real

A linha `RTC_DS3231 rtc;` instancia o objeto `rtc`, responsável por acessar as funções do módulo DS3231. Esse módulo mantém a contagem de data e hora mesmo quando o sistema é desligado, permitindo registrar eventos com precisão temporal.

2.3 Definição dos Pinos de Sensores e Atuadores

Os pinos são configurados de acordo com a função de cada componente:

- `pinoSensor1` e `pinoSensor2`: sensores de umidade do viveiro.
- `pinoValvula`: pino que aciona a válvula do viveiro.
- `pinoSensorH1`, `pinoSensorH2` e `pinoSensorH3`: sensores de umidade da horta.
- `pinoValvulaH`: pino que aciona a válvula solenoide da horta.
- `pinoFan`: pino responsável pelo controle do ventilador (fan) do sistema.

2.4 Leitura da Bateria e Controle do Relé de Comutação

Nesta parte, são definidos os parâmetros responsáveis pela leitura da tensão da bateria e pelo controle do relé:

- `R_DIV_TOP` e `R_DIV_BOT`: resistores que compõem o divisor de tensão, permitindo a medição da bateria sem exceder o limite de tensão do ADC do Arduino.
- `pinBatSense`: pino analógico que realiza a leitura da tensão do divisor.
- `pinRelayPower`: pino digital que aciona o relé de comutação de energia.
- `RELAY_ACTIVE_HIGH`: define a lógica de acionamento do relé (nível alto = ativado).
- `BAT_LOW` e `BAT_RESTORE`: limites de tensão que determinam quando o sistema deve desligar e religar cargas, protegendo a bateria contra descarga profunda.
- `lastSwitchMillis` e `MIN_SWITCH_INTERVAL`: controlam o intervalo mínimo entre comutações, prevenindo acionamentos repetitivos.

2.5 Níveis Lógicos de Acionamento

Os níveis lógicos de ativação dos dispositivos são definidos conforme o tipo de módulo utilizado:

- `VALVE_OPEN_LEVEL` e `VALVE_CLOSED_LEVEL`: determinam os estados de abertura e fechamento das válvulas.
- `FAN_ON_LEVEL` e `FAN_OFF_LEVEL`: estabelecem a lógica de funcionamento do ventilador.

Essas constantes permitem adaptar facilmente o código a diferentes tipos de módulos sem alterar a lógica principal do programa.

3 Parâmetros de Umidade, Controle de Irrigação e Funções Utilitárias

O trecho de código a seguir define os parâmetros de controle de umidade, os intervalos de tempo de leitura e irrigação, bem como variáveis de estado que registram o comportamento do sistema. Além disso, é apresentada uma função utilitária para formatação de data e hora.

```
// parametros de umidade e tempo
const int limiarSeco = 70;                  // quando iniciar rega (< limiarSeco)
const int TARGET_HUM = 75;                  // objetivo para parar a rega (>= TARGET_HUM)
const unsigned long CHECK_INTERVAL = 3600000UL; // 1 hora (mantido por compatibilidade)
const unsigned long IRR_POLL_INTERVAL = 5000UL; // enquanto irrigando, checar sensores a cada 5s
const unsigned long SAFETY_TIMEOUT = 10UL * 60UL * 1000UL;
const unsigned long SAFETY_TIMEOUT = 10UL * 60UL * 1000UL;
min (ajuste)
const unsigned long MIN_EVENT_INTERVAL = 2000UL;
intervalo (ms)
const int SAMPLE_COUNT = 5;
leitura
const unsigned long SAMPLE_DELAY = 120UL;
// display e SD
LiquidCrystal_I2C lcd(0x27, 20, 4);
const int addrHeaderFlag = 0;
const int addrRTCSetFlag = 1; // EEPROM flag para RTC ajustado
```

```
// estados Viveiro
bool irrigando = false;
unsigned long irrigationStartMillis = 0;
unsigned long lastCheckMillis = 0;
unsigned long lastIrrPollMillis = 0;
// estados Horta
bool irrigandoHorta = false;
unsigned long irrigationStartMillisH = 0;
unsigned long lastCheckMillisH = 0;
unsigned long lastIrrPollMillisH = 0;

String lastEvent = "";
unsigned long lastEventMillis = 0;

String timestamp(const DateTime &dt) {
  char b[20];
  snprintf(b, sizeof(b), "%04d/%02d/%02d %02d:%02d:%02d",
           dt.year(), dt.month(), dt.day(),
           dt.hour(), dt.minute(), dt.second());
  return String(b);}
```

2: Parâmetros de umidade e funções utilitárias

3.1 Parâmetros de Umidade e Controle de Irrigação

As variáveis declaradas nesta seção controlam o comportamento da irrigação automática, determinando quando iniciar e encerrar o processo, bem como o intervalo de monitoramento dos sensores:

- `limiarSeco`: define o limite inferior de umidade (<70%) a partir do qual o sistema inicia a irrigação.
- `TARGET_HUM`: representa o valor de umidade alvo (75%) no qual a irrigação deve ser interrompida.
- `CHECK_INTERVAL`: intervalo de verificação regular dos sensores (1 hora), utilizado para compatibilidade e monitoramento geral.
- `IRR_POLL_INTERVAL`: determina a frequência de leitura dos sensores durante o processo de irrigação (a cada 5 segundos).
- `SAFETY_TIMEOUT`: tempo máximo permitido para a irrigação (10 minutos), funcionando como medida de segurança para evitar sobrecarga de água.

- `MIN_EVENT_INTERVAL`: intervalo mínimo entre eventos de log, prevenindo registros duplicados em curto prazo.
- `SAMPLE_COUNT` e `SAMPLE_DELAY`: controlam o número de amostras rápidas e o atraso entre elas, permitindo leituras mais estáveis e filtradas dos sensores analógicos.

3.2 Configuração do Display e Memória EEPROM

O display LCD é configurado utilizando a biblioteca `LiquidCrystal_I2C`, com endereço I2C `0x27` e dimensões de 20 colunas por 4 linhas. Essa configuração permite exibir informações de status, níveis de umidade e mensagens de sistema.

As variáveis `addrHeaderFlag` e `addrRTCSetFlag` representam endereços na memória EEPROM:

- `addrHeaderFlag`: indica se o cabeçalho do arquivo no cartão SD já foi gravado.
- `addrRTCSetFlag`: armazena um indicador de que o relógio de tempo real (RTC) foi ajustado corretamente.

3.3 Estados do Sistema

Foram criadas variáveis de estado tanto para o viveiro quanto para a horta, permitindo o controle independente de cada setor:

- `irrigando` e `irrigandoHorta`: indicam se a irrigação está em andamento.
- `irrigationStartMillis` e `irrigationStartMillisH`: armazenam o momento de início da irrigação (em milissegundos).
- `lastCheckMillis` e `lastCheckMillisH`: registram o último instante de verificação dos sensores.
- `lastIrrPollMillis` e `lastIrrPollMillisH`: indicam o último momento em que os sensores foram lidos durante a irrigação.

Além disso, a variável `lastEvent` armazena a descrição textual do último evento registrado, enquanto `lastEventMillis` guarda o tempo em que esse evento ocorreu. Essas informações são utilizadas para fins de log e monitoramento do histórico de ações do sistema.

3.4 Função Utilitária `timestamp`

A função `timestamp` é responsável por converter o objeto `DateTime` (fornecido pela biblioteca `RTClib`) em uma string legível no formato `YYYY/MM/DD HH:MM:SS`. Isso permite a gravação e exibição de registros com data e hora completas, facilitando o acompanhamento cronológico dos eventos registrados no cartão SD ou mostrados no display LCD.

4 Função de Registro Seguro de Eventos `safeLog`

A função a seguir tem como objetivo registrar eventos e leituras do sistema em um arquivo `log.csv` armazenado no cartão SD. Ela foi desenvolvida de forma a evitar duplicação de registros e a garantir integridade dos dados, mesmo em situações de falha momentânea no sistema.Assim garantindo a maior confiabilidade dos dados, a função atua como o ponto centralizado para o registro de todos os eventos e dados operacionais. Sua implementação garante a integridade do processo de logging e a uniformidade dos dados.

```
1  void safeLog(const String &ts, const char* origem, int u1, int u2, int
       u3, int u4, const char* ev) {
2    unsigned long nowMs = millis();
3    String evStr = String(ev);
4    if (evStr == lastEvent && (nowMs - lastEventMillis) <
       MIN_EVENT_INTERVAL) {
5      Serial.println("Ignorado log duplicado: " + evStr);
6      return;
7    }
8    File f = SD.open("log.csv", FILE_WRITE);
9    if (f) {
10     f.print(ts); f.print(',');
11     f.print(origem); f.print(',');
12     if (u1 >= 0) f.print(u1);
13     f.print(',');
14     if (u2 >= 0) f.print(u2);
15     f.print(',');
16     if (u3 >= 0) f.print(u3);
17     f.print(',');
18     if (u4 >= 0) f.print(u4);
19     f.print(',');
20     f.println(ev);
21     f.close();
```

```
22      Serial.println(String(ev) + " (" + String(origem) + ") gravado: " +
            ts);
23    } else {
24      Serial.println("Falha ao abrir log.csv para escrita");
25    }
26    lastEvent = evStr;
27    lastEventMillis = nowMs;
28  }
```

3: Função de registro seguro `safeLog`

4.1 Descrição da Função

A função `safeLog` centraliza o processo de registro de informações no sistema, permitindo que dados provenientes tanto do *viveiro* quanto da *horta* sejam armazenados em um único arquivo de log no formato CSV (`log.csv`). Seu design unificado simplifica a análise posterior dos dados, além de facilitar a manutenção do sistema.

4.2 Parâmetros da Função

- `ts`: representa o carimbo de data e hora (timestamp) no formato `YYYY/MM/DD HH:MM:SS`, normalmente obtido pela função `timestamp()`.
- `origem`: indica a origem do evento (`"viveiro"`, `"horta"` ou `"system"`), permitindo identificar o setor do sistema que gerou o registro.
- `u1, u2, u3, u4`: são valores numéricos genéricos utilizados para registrar medições ou variáveis relevantes, como leituras de sensores de umidade ou temperatura. Valores negativos são interpretados como campos vazios e não são escritos no CSV.
- `ev`: string descritiva do evento a ser registrado (por exemplo, `"Início de irrigação"` ou `"Umidade normalizada"`).

4.3 Mecanismo de Prevenção de Logs Duplicados

A função implementa um sistema de verificação para evitar o registro repetido de eventos idênticos em um curto intervalo de tempo. Isso é feito comparando o evento atual (`evStr`) com o último evento registrado (`lastEvent`) e verificando se o tempo decorrido desde o último log (`nowMs - lastEventMillis`) é inferior ao intervalo mínimo `MIN_EVENT_INTERVAL`. Caso ambas as condições sejam verdadeiras, o evento é ignorado e uma mensagem é exibida via porta serial:

```
Ignorado log duplicado:  <evento>
```

Esse controle é essencial para evitar redundância no arquivo de log, especialmente quando o sistema é executado em ciclos rápidos de verificação.

4.4 Processo de Escrita no Cartão SD

Quando a verificação de duplicidade é superada, a função tenta abrir o arquivo `log.csv` em modo de escrita (`FILE_WRITE`). Se a abertura for bem-sucedida:

1. O timestamp, a origem e os valores numéricos são escritos sequencialmente, separados por vírgulas.
2. Campos com valores negativos são deixados vazios, preservando a estrutura do CSV.
3. O evento textual é então adicionado ao final da linha, seguido de uma quebra de linha.
4. O arquivo é fechado em seguida, garantindo que os dados fiquem gravados de forma segura.

Em caso de falha na abertura do arquivo, é exibida uma mensagem de erro via `Serial`:

```
Falha ao abrir log.csv para escrita
```

4.5 Atualização dos Registros de Estado

Após a gravação, a função atualiza as variáveis globais:

- `lastEvent`: armazena o nome do último evento registrado.
- `lastEventMillis`: guarda o instante em que a gravação foi realizada.

Essas informações são usadas nas próximas chamadas da função para garantir a consistência do histórico de eventos e impedir duplicações desnecessárias.

4.6 Importância no Contexto do Sistema

A `safeLog` é um componente essencial do sistema de automação, pois integra a coleta de dados e a rastreabilidade das ações executadas. Com o uso dessa função, é possível:

- Manter um histórico organizado e cronológico de eventos.

- Facilitar auditorias e análises posteriores de desempenho da irrigação.
- Evitar a perda ou repetição de dados, mesmo em cenários de interrupção temporária.

Assim, a função contribui diretamente para a confiabilidade e transparência do sistema automatizado de irrigação.

5 Funções de Leitura dos Sensores e Controle das Válvulas

As funções apresentadas a seguir são responsáveis pela leitura dos sensores de umidade do solo e pelo controle das válvulas de irrigação tanto do viveiro quanto da horta. Essas rotinas constituem a base da operação do sistema, garantindo medições estáveis e ações precisas de irrigação.

```
1
2  void readSensorsAvgTwo(int pin1, int pin2, int &u1_out, int &u2_out,
       float &med_out) {
3    long sum1 = 0;
4    long sum2 = 0;
5    for (int i = 0; i < SAMPLE_COUNT; ++i) {
6      wdt_reset();
7      int r1 = analogRead(pin1);
8      int r2 = analogRead(pin2);
9      sum1 += map(r1, 1023, 0, 0, 100);
10     sum2 += map(r2, 1023, 0, 0, 100);
11     delay(SAMPLE_DELAY);
12   }
13   int u1 = sum1 / SAMPLE_COUNT;
14   int u2 = sum2 / SAMPLE_COUNT;
15   float med = (u1 + u2) / 2.0;
16   u1_out = u1;
17   u2_out = u2;
18   med_out = med;
19 }
20
21
22 void readSensorsAvgThree(int p1, int p2, int p3, int &s1_out, int &
       s2_out, int &s3_out, float &med_out) {
23   long sum1 = 0;
24   long sum2 = 0;
```

```
  long sum3 = 0;
  for (int i = 0; i < SAMPLE_COUNT; ++i) {
    wdt_reset();
    int r1 = analogRead(p1);
    int r2 = analogRead(p2);
    int r3 = analogRead(p3);
    sum1 += map(r1, 1023, 0, 0, 100);
    sum2 += map(r2, 1023, 0, 0, 100);
    sum3 += map(r3, 1023, 0, 0, 100);
    delay(SAMPLE_DELAY);
  }
  int s1 = sum1 / SAMPLE_COUNT;
  int s2 = sum2 / SAMPLE_COUNT;
  int s3 = sum3 / SAMPLE_COUNT;
  float med = (s1 + s2 + s3) / 3.0;
  s1_out = s1; s2_out = s2; s3_out = s3; med_out = med;
}


void openValve() {//abre e fecha a valvula do viveiro
  digitalWrite(pinoValvula, VALVE_OPEN_LEVEL);
}
void closeValve() {
  digitalWrite(pinoValvula, VALVE_CLOSED_LEVEL);
}


void openValveHorta() {//abre e fecha a valvula da horta
  digitalWrite(pinoValvulaH, VALVE_OPEN_LEVEL);
}
void closeValveHorta() {
  digitalWrite(pinoValvulaH, VALVE_CLOSED_LEVEL);
}
```

4: Funções de leitura dos sensores e controle das válvulas

5.1 Leitura dos Sensores de Umidade

As funções `readSensorsAvgTwo` e `readSensorsAvgThree` realizam a leitura dos sensores de umidade analógicos conectados ao microcontrolador, correspondendo respectivamente ao viveiro (2 sensores) e à horta (3 sensores). Essas leituras são feitas múltiplas vezes e posteriormente suavizadas por meio da média aritmética, o que reduz o ruído elétrico e aumenta a confiabilidade das medições.

5.1.1 Processo de Amostragem

Cada função realiza um laço de repetição (`for`) com `SAMPLE_COUNT` iterações, definido previamente como o número de amostras por leitura. Durante cada iteração:

1. O microcontrolador reinicia o temporizador de segurança por meio da função `wdt_reset()`, evitando que o *watchdog timer* reinicie o sistema durante o processo.
2. É realizada a leitura analógica dos sensores via `analogRead()`.
3. Os valores lidos, originalmente variando entre 0 e 1023, são convertidos para uma escala percentual (0 a 100) com a função `map()`, o que facilita a interpretação da umidade.
4. Cada valor convertido é acumulado em uma variável de soma (`sum1`, `sum2`, `sum3`).
5. Um pequeno atraso (`delay(SAMPLE_DELAY)`) é aplicado entre as leituras para garantir estabilidade no sinal.

Ao final do processo, as médias de cada sensor são calculadas e armazenadas nas variáveis de saída passadas por referência. Além disso, uma média geral (`med_out`) é calculada com base na média das leituras individuais, representando a umidade média total da área monitorada.

5.2 Conversão e Normalização dos Dados

O uso da função `map()` transforma a leitura do conversor analógico-digital (ADC) de 10 bits em uma escala intuitiva, onde 0 representa solo completamente seco e 100 indica umidade máxima. Isso simplifica a comparação com os limiares de irrigação (`limiarSeco` e `TARGET_HUM`) definidos anteriormente.

5.3 Controle das Válvulas de Irrigação

As funções `openValve()`, `closeValve()`, `openValveHorta()` e `closeValveHorta()` realizam o controle direto das válvulas solenoides responsáveis pela liberação ou bloqueio do fluxo de água. Essas funções enviam sinais digitais de acordo com o nível lógico configurado nas constantes `VALVE_OPEN_LEVEL` e `VALVE_CLOSED_LEVEL`, que podem variar dependendo do tipo de módulo utilizado.

5.3.1 Operação e Segurança

O controle direto das válvulas permite ao sistema acionar a irrigação apenas quando os sensores indicam níveis de umidade abaixo do valor mínimo desejado. A abstração em funções separadas simplifica a leitura do código e permite futuras alterações de hardware sem necessidade de reescrever a lógica de irrigação principal.

5.4 Importância no Contexto do Sistema

Essas rotinas desempenham papel fundamental na automação do sistema, pois unem a precisão da coleta de dados com a confiabilidade do acionamento das válvulas. Com elas, o sistema consegue:

- Efetuar leituras consistentes e filtradas de umidade do solo;
- Converter os dados para uma escala percentual compreensível;
- Acionar de forma controlada as válvulas de irrigação conforme a necessidade;
- Garantir estabilidade e segurança durante todo o processo de leitura e controle.

O conjunto dessas funções garante a eficiência e a autonomia da irrigação automatizada, promovendo economia de água e manutenção ideal das plantas do viveiro e da horta.

6 Leitura e Controle do Sistema de Alimentação

O conjunto de funções a seguir é responsável pela leitura do nível de tensão da bateria e pela comutação automática entre a alimentação via bateria e o carregador. Esse mecanismo garante a operação contínua do sistema e evita o descarregamento profundo da bateria, preservando sua vida útil.

```
1  // ---------- leitura e controle da bateria ----------
2  float readBatteryVoltage() {
3    const int SAMPLES = 10;
4    long sum = 0;
5    for (int i = 0; i < SAMPLES; ++i) {
6      sum += analogRead(pinBatSense);
7      delay(5);
8    }
9    float adc = (float)sum / (float)SAMPLES;
10   float vout = adc * (5.0 / 1023.0);
11   float vin = vout * ((R_DIV_TOP + R_DIV_BOT) / R_DIV_BOT);
```

```
  return vin;
}
// seleciona bateria (true) ou carregador (false)
void selectBatterySource(bool useBattery) {
  if (RELAY_ACTIVE_HIGH) {
    digitalWrite(pinRelayPower, useBattery ? HIGH : LOW);
  } else {
    digitalWrite(pinRelayPower, useBattery ? LOW : HIGH);
  }
}

void checkBatteryAndSwitch() {
  static bool usingBattery = false; // estado corrente
  unsigned long now = millis();
  if (now - lastSwitchMillis < MIN_SWITCH_INTERVAL) return;

  float vbatt = readBatteryVoltage();
  Serial.print("Vbat: "); Serial.println(vbatt);

  if (usingBattery) {
    if (vbatt <= BAT_LOW) {
      selectBatterySource(false); // vai para carregador
      usingBattery = false;
      lastSwitchMillis = now;
      safeLog(timestamp(rtc.now()), "system", -1, -1, -1, -1, "
          power_switch_to_charger");
      Serial.println("Switch -> CHARGER (bateria baixa)");
    }
  } else {
    if (vbatt >= BAT_RESTORE) {
      selectBatterySource(true); // volta pra bateria
      usingBattery = true;
      lastSwitchMillis = now;
      safeLog(timestamp(rtc.now()), "system", -1, -1, -1, -1, "
          power_switch_to_battery");
      Serial.println("Switch -> BATTERY (bateria ok)");
    }
  }
}
```

5: Funções de leitura e controle da bateria

6.1 Função `readBatteryVoltage()`

A função `readBatteryVoltage()` é responsável por realizar a leitura da tensão da bateria utilizando o pino analógico `A6`, conectado a um divisor resistivo composto pelos resistores `R_DIV_TOP` e `R_DIV_BOT`. Esse divisor é necessário, pois a tensão da bateria (superior a 5V) precisa ser reduzida para o intervalo seguro de medição do conversor analógico-digital (ADC) do microcontrolador.

O processo de medição ocorre em etapas:

1. São realizadas múltiplas leituras consecutivas (`SAMPLES = 10`) no pino analógico.
2. As leituras são somadas e a média é calculada para minimizar ruídos elétricos.
3. O valor médio é convertido para tensão real (`vout`) com base na referência de 5V e na resolução de 10 bits do ADC.
4. Em seguida, a tensão da bateria (`vin`) é calculada aplicando a razão entre os resistores do divisor.

Essa abordagem proporciona uma leitura precisa e estável do nível de carga da bateria, sendo fundamental para a lógica de comutação descrita a seguir.

6.2 Função `selectBatterySource()`

A função `selectBatterySource()` realiza a comutação entre as fontes de energia do sistema: **bateria** (quando `useBattery = true`) e **carregador** (quando `useBattery = false`). A lógica implementada considera o tipo de relé utilizado no hardware, podendo ser *ativo em nível alto* ou *ativo em nível baixo*, conforme definido na constante `RELAY_ACTIVE_HIGH`.

Dessa forma, o código torna-se independente da configuração física dos módulos de relé, permitindo reutilização e adaptação do sistema a diferentes montagens de circuito.

6.3 Função `checkBatteryAndSwitch()`

A função `checkBatteryAndSwitch()` é projetada para ser executada periodicamente (por exemplo, dentro da função `loop()`). Seu objetivo é verificar o estado atual da bateria e decidir automaticamente se o sistema deve operar em modo bateria ou conectado ao carregador.

6.3.1 Etapas de Execução

1. A variável estática `usingBattery` mantém o estado atual do sistema (alimentação pela bateria ou pelo carregador).
2. O tempo desde a última comutação é verificado para evitar trocas rápidas, respeitando o intervalo mínimo definido por `MIN_SWITCH_INTERVAL`.
3. A tensão da bateria é obtida pela função `readBatteryVoltage()` e exibida via comunicação serial.
4. Se o sistema está utilizando a bateria e a tensão cair abaixo de `BAT_LOW`, o relé é acionado para mudar a alimentação para o carregador.
5. Caso o sistema esteja no carregador e a tensão ultrapasse `BAT_RESTORE`, a alimentação é revertida novamente para a bateria.

Cada comutação de fonte é registrada no log por meio da função `safeLog()`, que armazena o evento correspondente no cartão SD (`power_switch_to_charger` ou `power_switch_to_battery`), garantindo rastreabilidade completa das operações do sistema.

6.4 Importância do Controle de Alimentação

O controle automático de comutação é fundamental em sistemas autônomos baseados em energia solar ou híbrida. Ele impede que a bateria seja descarregada além do limite seguro, protegendo o conjunto de células e mantendo a operação contínua do sistema.

Além disso, a integração com o registro de eventos possibilita a análise de desempenho energético, permitindo identificar padrões de carga e descarga, bem como ajustar o dimensionamento da fonte de alimentação quando necessário.

7 Funções Agendadas de Verificação e Irrigação

As funções apresentadas a seguir implementam o núcleo da rotina de automação, realizando verificações periódicas dos níveis de umidade do solo e acionando a irrigação sempre que necessário. Essas funções são chamadas de forma agendada, com base em intervalos temporais definidos no código principal (`loop()`), permitindo o monitoramento contínuo das condições do viveiro e da horta.

```
1
2 void performViveiroCheck(unsigned long nowMillis, const DateTime &now)
    {
```

```
  if (!irrigando) {
    int u1, u2; float med;
    readSensorsAvgTwo(pinoSensor1, pinoSensor2, u1, u2, med);

    Serial.print("Viveiro Scheduled Check: u1="); Serial.print(u1);
    Serial.print(" u2="); Serial.print(u2);
    Serial.print(" med="); Serial.println(med);

    if (med < limiarSeco) {
      irrigando = true;
      irrigationStartMillis = nowMillis;
      lastIrrPollMillis = nowMillis;
      openValve();
      safeLog(timestamp(now), "viveiro", u1, u2, -1, -1, "
          valve_open_viveiro");
      Serial.println("Viveiro: Iniciando rega.");
    } else {
      Serial.println("Viveiro: Solo ok - sem rega neste check.");
    }
  }
}


void performHortaCheck(unsigned long nowMillis, const DateTime &now) {
  if (!irrigandoHorta) {
    int hs1, hs2, hs3; float hmed;
    readSensorsAvgThree(pinoSensorH1, pinoSensorH2, pinoSensorH3, hs1,
        hs2, hs3, hmed);

    Serial.print("Horta Scheduled Check: h1="); Serial.print(hs1);
    Serial.print(" h2="); Serial.print(hs2);
    Serial.print(" h3="); Serial.print(hs3);
    Serial.print(" med="); Serial.println(hmed);

    if (hmed < limiarSeco) {
      irrigandoHorta = true;
      irrigationStartMillisH = nowMillis;
      lastIrrPollMillisH = nowMillis;
      openValveHorta();
      safeLog(timestamp(now), "horta", hs1, hs2, hs3, -1, "
          valve_open_horta");
      Serial.println("Horta: Iniciando rega.");
    } else {
      Serial.println("Horta: Solo ok - sem rega neste check.");
    }
```

```
45      }
46  }
```

6: Funções agendadas de verificação do viveiro e da horta

7.1 Função `performViveiroCheck()`

A função `performViveiroCheck()` é responsável por avaliar a umidade do solo no setor do viveiro. Ela utiliza dois sensores (`pinoSensor1` e `pinoSensor2`) e calcula a média das leituras para determinar o nível médio de umidade (`med`).

7.1.1 Etapas de execução:

1. Verifica se o sistema não está atualmente irrigando (`!irrigando`).
2. Realiza a leitura dos sensores por meio da função auxiliar `readSensorsAvgTwo()`, obtendo valores individuais e a média.
3. Exibe as leituras no monitor serial para fins de depuração e registro.
4. Caso a média (`med`) esteja abaixo do limite definido por `limiarSeco`, o sistema inicia o processo de irrigação:
    - Ativa o estado `irrigando = true`;
    - Registra o instante de início da irrigação;
    - Abre a válvula de água com `openValve()`;
    - Registra o evento no log por meio da função `safeLog()`.
5. Se o solo estiver dentro do nível adequado, o sistema apenas informa que não há necessidade de irrigação naquele momento.

Esse comportamento garante que o viveiro receba irrigação apenas quando realmente necessário, evitando desperdício de água e otimizando o uso dos recursos energéticos.

7.2 Função `performHortaCheck()`

A função `performHortaCheck()` possui estrutura e lógica semelhantes à anterior, porém voltada à área da horta, onde são utilizados três sensores (`pinoSensorH1`, `pinoSensorH2` e `pinoSensorH3`). A leitura é feita através da função `readSensorsAvgThree()`, que fornece os valores individuais e a média (`hmed`).

7.2.1 Etapas de execução:

1. Verifica se não há irrigação em andamento na horta (`!irrigandoHorta`).
2. Realiza as medições e exibe os resultados no monitor serial.
3. Caso a média de umidade esteja abaixo de `limiarSeco`, inicia automaticamente a irrigação da horta, abrindo a válvula específica com `openValveHorta()`.
4. Todos os eventos são registrados no log com as leituras dos sensores e o identificador de ação `valve_open_horta`.

7.3 Análise Geral do Comportamento das Funções

Ambas as rotinas funcionam de maneira independente, o que permite que o sistema controle setores distintos do cultivo de forma simultânea, porém segura. Cada setor possui suas próprias variáveis de controle, tempos de irrigação e verificações periódicas.

O uso de médias ponderadas e limiares de umidade evita falsos acionamentos, que poderiam ocorrer por leituras momentâneas incorretas ou ruído elétrico. Além disso, o registro dos eventos via `safeLog()` proporciona rastreabilidade completa do histórico de irrigações, contribuindo para análises posteriores de eficiência hídrica e comportamento do solo.

7.4 Importância da Automação Periódica

O agendamento automático das leituras e irrigações é um dos aspectos mais importantes da automação agrícola. Ele permite que o sistema opere de forma autônoma, ajustando-se às condições reais do solo e eliminando a necessidade de supervisão constante. Isso representa não apenas um avanço tecnológico, mas também um impacto direto na sustentabilidade e na gestão inteligente dos recursos naturais.

8 Configuração Inicial e Processamento de Comandos Seriais

O trecho a seguir contém duas funções centrais do sistema embarcado: `setup()` e `processSerialCommands()`. A função `setup()` é executada uma única vez ao iniciar o microcontrolador e tem a função de configurar os periféricos, inicializar os sensores e preparar o ambiente de execução. Já a função `processSerialCommands()` permite a interação direta com o sistema via porta serial, possibilitando o envio de comandos manuais para depuração, teste e ajuste do relógio de tempo real (RTC).

```
// ---------- setup ----------
void setup() {
  // watchdog (reinicia se travar)
  wdt_enable(WDTO_8S);

  Serial.begin(9600);
  Wire.begin();

  // RTC
  if (!rtc.begin()) {
    Serial.println("Erro: RTC n o encontrado");
    while (1) { /* trava pra indicar erro */ }
  }

  // ajustar RTC apenas se nunca tiver sido configurado antes
  bool needSet = false;
  if (EEPROM.read(addrRTCSetFlag) != 1) needSet = true;

  if (needSet) {
    rtc.adjust(DateTime(__DATE__, __TIME__));
    EEPROM.write(addrRTCSetFlag, 1);
    Serial.println("RTC ajustado para hora de upload (compile-time).");
  } else {
    Serial.println("RTC j configurado previamente. Mantendo hora.");
  }

  // SD: cabe alho apenas uma vez (unificado: origem + 4 sensores)
  pinMode(chipSelect, OUTPUT);
  if (SD.begin(chipSelect)) {
    File f = SD.open("log.csv", FILE_READ);
    bool needHeader = (!f || f.size() == 0);
    if (f) f.close();
    if (needHeader) {
      File h = SD.open("log.csv", FILE_WRITE);
      if (h) {
        h.println("datetime,origem,umid1,umid2,umid3,umid4,evento");
        h.close();
        EEPROM.write(addrHeaderFlag, 1);
        Serial.println("Cabe alho gravado em log.csv");
      }
    } else {
      Serial.println("log.csv j existe");
    }
  } else {
```

```
45      Serial.println("Falha ao inicializar SD");
46    }
47
48    // LCD
49    lcd.init();
50    lcd.backlight();
51
52    // pinos de sa da
53    pinMode(pinoValvula, OUTPUT);
54    closeValve();
55    pinMode(pinoValvulaH, OUTPUT);
56    closeValveHorta();
57    pinMode(pinoFan, OUTPUT);
58    digitalWrite(pinoFan, FAN_OFF_LEVEL);
59
60    // pinos para controle de energia
61    pinMode(pinRelayPower, OUTPUT);
62    // inicial: assume preferencia ao carregador (rel  OFF -> NC ->
          charger)
63    selectBatterySource(false);
64
65    // inicializa temporizadores
66    lastCheckMillis = millis();
67    lastIrrPollMillis = millis();
68    lastCheckMillisH = millis();
69    lastIrrPollMillisH = millis();
70
71    Serial.println("Comandos Serial: 'SET YYYY-MM-DD HH:MM:SS' | 'T' | '
          GET' | 'TESTLOG'");
72  }
73
74  // ---------- Serial: processa comando de entrada ----------
75  void processSerialCommands() {
76    if (!Serial.available()) return;
77    String line = Serial.readStringUntil('\n');
78    line.trim();
79    if (line.length() == 0) return;
80
81    if (line.equalsIgnoreCase("T")) {//ajuste manual apenas se necessario
82      rtc.adjust(DateTime(__DATE__, __TIME__));
83      EEPROM.write(addrRTCSetFlag, 1);
84      Serial.println("RTC ajustado manualmente para hora de compilacao/
            upload.");
85      return;
86    }
```

```
87
88  if (line.equalsIgnoreCase("GET")) {
89    DateTime now = rtc.now();
90    Serial.println("RTC: " + timestamp(now));
91    return;
92  }
93
94
95  if (line.equalsIgnoreCase("TESTLOG")) {
96    int v1, v2; float vm;
97    readSensorsAvgTwo(pinoSensor1, pinoSensor2, v1, v2, vm);
98    safeLog(timestamp(rtc.now()), "viveiro", v1, v2, -1, -1, "
        manual_test_viveiro");
99
100   int h1, h2, h3; float hm;
101   readSensorsAvgThree(pinoSensorH1, pinoSensorH2, pinoSensorH3, h1,
        h2, h3, hm);
102   safeLog(timestamp(rtc.now()), "horta", h1, h2, h3, -1, "
        manual_test_horta");
103
104   Serial.println("Log manual (viveiro + horta) gravado no SD.");
105   return;
106 }
107
108 // SET YYYY-MM-DD HH:MM:SS
109 if (line.startsWith("SET ")) {
110   String payload = line.substring(4);
111   int y=0, M=0, d=0, h=0, m=0, s=0;
112   if (sscanf(payload.c_str(), "%d-%d-%d %d:%d:%d", &y, &M, &d, &h, &m
        , &s) == 6) {
113     if (y>=2000 && M>=1 && M<=12 && d>=1 && d<=31 && h>=0 && h<24 &&
          m>=0 && m<60 && s>=0 && s<60) {
114       rtc.adjust(DateTime(y, M, d, h, m, s));
115       EEPROM.write(addrRTCSetFlag, 1);
116       Serial.print("RTC ajustado para: ");
117       Serial.println(payload);
118     } else {
119       Serial.println("Valores fora do intervalo valido.");
120     }
121   } else {
122     Serial.println("Formato inv lido. Use: SET YYYY-MM-DD HH:MM:SS")
          ;
123   }
124   return;
125 }
```

```
126
127   Serial.println("Comando desconhecido.");
128 }
```

7: Funções de inicialização e processamento serial

8.1 Descrição da Função `setup()`

A função `setup()` executa as rotinas de configuração inicial do sistema. Entre suas principais funções, destacam-se:

- Ativação do **watchdog timer** (`wdt_enable(WDTO_8S)`), responsável por reiniciar o sistema em caso de travamento;
- Inicialização da comunicação serial (`Serial.begin(9600)`) e do barramento I2C (`Wire.begin()`);
- Verificação e configuração inicial do módulo de tempo real (RTC), armazenando o estado de configuração em memória EEPROM;
- Inicialização do módulo de armazenamento SD, incluindo a criação automática de um arquivo de log (`log.csv`) com cabeçalho padrão, caso ainda não exista;
- Configuração e ativação do display LCD com retroiluminação;
- Definição dos pinos de saída para controle das válvulas de irrigação, ventilador e relé de energia;
- Inicialização dos temporizadores utilizados nas verificações periódicas de umidade.

Essas configurações garantem que, ao ser energizado, o sistema esteja completamente funcional, com todos os módulos corretamente inicializados e sincronizados.

8.2 Descrição da Função `processSerialCommands()`

A função `processSerialCommands()` permite o envio de comandos via porta serial, funcionando como uma interface de controle manual. Os principais comandos implementados são:

- `T`: ajusta o RTC para o horário de compilação;
- `GET`: exibe a data e hora atual do RTC no monitor serial;

- `TESTLOG`: força a gravação de um registro manual de umidade tanto do viveiro quanto da horta no cartão SD;
- `SET YYYY-MM-DD HH:MM:SS`: ajusta manualmente o RTC para uma data e hora específicas;

A função também trata erros de entrada e garante que apenas comandos válidos sejam executados, exibindo mensagens descritivas no monitor serial.

9 Função `loop()`: Execução Contínua do Sistema

A função `loop()` representa o núcleo do funcionamento contínuo do sistema embarcado. Nela, são executadas as tarefas de monitoramento, controle e atualização dos dispositivos em tempo real. Assim como em todos os projetos Arduino, o `loop()` é executado de forma cíclica e indefinida, garantindo o funcionamento ininterrupto do sistema.

9.1 Reinicialização e Comandos Seriais

Logo no início do `loop()`, o comando `wdt_reset()` é responsável por reinicializar o *watchdog timer*, evitando que o microcontrolador seja reiniciado indevidamente. Em seguida, a função `processSerialCommands()` verifica se há comandos recebidos via porta serial, possibilitando o ajuste de hora, leitura de dados e testes de gravação manual no cartão SD.

```
// ---------- loop ----------
void loop() {
  // reseta watchdog
  wdt_reset();

  // processa comandos Serial (ajuste de RTC, GET, TESTLOG)
  processSerialCommands();

  // manter backlight
  lcd.backlight();

  DateTime now = rtc.now();
  int h = now.hour();
  String ts = timestamp(now);
```

8: Reinicialização e Comandos Seriais

9.2 Controle da Ventoinha

Com base na hora atual obtida pelo módulo RTC, o sistema realiza o controle da ventoinha (fan). O ventilador é ativado ou desativado automaticamente conforme o horário do dia, permanecendo ligado entre 10h e 18h, de modo a garantir a ventilação adequada durante o período mais quente.

```
    // Controle da fan (10  18h  )
if (h >= 10 && h < 18) digitalWrite(pinoFan, FAN_OFF_LEVEL);
else digitalWrite(pinoFan, FAN_ON_LEVEL);

unsigned long nowMillis = millis();
```

9: Controle do Fan

9.3 Monitoramento da Bateria

A função `checkBatteryAndSwitch()` é chamada para verificar o estado da bateria e realizar a comutação automática entre as fontes de energia (bateria ou carregador), assegurando o funcionamento contínuo mesmo em variações na alimentação elétrica.

9.4 Agendamento de Irrigação

O sistema realiza agendamentos automáticos de irrigação baseados no relógio RTC:

- **Viveiro**: a cada hora cheia (minuto 0);
- **Horta**: a cada meia hora (minuto 30).

Esses agendamentos são controlados por variáveis de índice de minuto, evitando repetições desnecessárias e garantindo que cada ciclo ocorra apenas uma vez no tempo previsto.

```
  // ---------- Monitor de bateria e comuta  o ----------
checkBatteryAndSwitch();

// ---------- Agendamento via RTC: Viveiro no minuto 0, Horta no
    minuto 30 ----------
static long lastViveiroMinuteIndex = -1;
static long lastHortaMinuteIndex = -1;
uint32_t minuteIndex = now.unixtime() / 60UL;

if (now.minute() == 0 && minuteIndex != lastViveiroMinuteIndex) {
```

```
    lastViveiroMinuteIndex = minuteIndex;
    performViveiroCheck(nowMillis, now);
  }

  if (now.minute() == 30 && minuteIndex != lastHortaMinuteIndex) {
    lastHortaMinuteIndex = minuteIndex;
    performHortaCheck(nowMillis, now);
  }
```

10: Monitoramento da Bateria

9.5 Monitoramento Durante a Irrigação

Durante o processo de irrigação, o sistema realiza leituras periódicas dos sensores de umidade para o viveiro e para a horta. Caso a umidade média (`med` ou `hmed`) alcance o valor-alvo definido por `TARGET_HUM`, a válvula é automaticamente fechada e o evento é registrado no cartão SD com um rótulo informativo, como `valve_close_target_viveiro`. Além disso, há um mecanismo de segurança denominado `SAFETY_TIMEOUT`, que interrompe a irrigação após um tempo máximo para evitar falhas de travamento ou vazamentos.

```
    / Irrigation polling (enquanto irrigando)
  // Viveiro polling
  if (irrigando) {
    if (nowMillis - lastIrrPollMillis >= IRR_POLL_INTERVAL) {
      lastIrrPollMillis = nowMillis;
      int u1, u2; float med;
      readSensorsAvgTwo(pinoSensor1, pinoSensor2, u1, u2, med);

      Serial.print("Viveiro Irriga poll: u1="); Serial.print(u1);
      Serial.print(" u2="); Serial.print(u2);
      Serial.print(" med="); Serial.println(med);

      if (med >= TARGET_HUM) {
        closeValve();
        safeLog(timestamp(rtc.now()), "viveiro", u1, u2, -1, -1, "
            valve_close_target_viveiro");
        irrigando = false;
```

```
17          Serial.println("Viveiro: Parou rega: alvo de umidade atingido."
                );
18        } else if (SAFETY_TIMEOUT > 0 && (nowMillis -
              irrigationStartMillis) >= SAFETY_TIMEOUT) {
19          closeValve();
20          safeLog(timestamp(rtc.now()), "viveiro", u1, u2, -1, -1, "
                valve_close_timeout_viveiro");
21          irrigando = false;
22          Serial.println("Viveiro: Parou rega: safety timeout.");
23        }
24      }
25    }
26
27    // Horta polling
28    if (irrigandoHorta) {
29      if (nowMillis - lastIrrPollMillisH >= IRR_POLL_INTERVAL) {
30        lastIrrPollMillisH = nowMillis;
31        int hs1, hs2, hs3; float hmed;
32        readSensorsAvgThree(pinoSensorH1, pinoSensorH2, pinoSensorH3, hs1
              , hs2, hs3, hmed);
33
34        Serial.print("Horta Irriga poll: h1="); Serial.print(hs1);
35        Serial.print(" h2="); Serial.print(hs2);
36        Serial.print(" h3="); Serial.print(hs3);
37        Serial.print(" med="); Serial.println(hmed);
38
39        if (hmed >= TARGET_HUM) {
40          closeValveHorta();
41          safeLog(timestamp(rtc.now()), "horta", hs1, hs2, hs3, -1, "
                valve_close_target_horta");
42          irrigandoHorta = false;
43          Serial.println("Horta: Parou rega: alvo de umidade atingido.");
44        } else if (SAFETY_TIMEOUT > 0 && (nowMillis -
              irrigationStartMillisH) >= SAFETY_TIMEOUT) {
45          closeValveHorta();
46          safeLog(timestamp(rtc.now()), "horta", hs1, hs2, hs3, -1, "
                valve_close_timeout_horta");
47          irrigandoHorta = false;
48          Serial.println("Horta: Parou rega: safety timeout.");
49        }
50      }
51    }
```

11: Monitoramento durante a Irrigação

9.6 Atualização do Display LCD

A cada iteração, o sistema atualiza as informações exibidas no display LCD:

- Linha 0: Umidade média do viveiro;
- Linha 1: Umidade média da horta;
- Linha 2: Data e hora atuais;
- Linha 3: Estado atual do sistema (`Regando Viveiro`, `Regando Horta` ou `Solo OK`).

Essas informações são formatadas e exibidas de forma organizada, facilitando o acompanhamento visual do sistema em tempo real.

```
// =========================
  // Atualiza LCD (mostra m dias e status)
  // =========================
  int v1_disp, v2_disp; float vmed_disp;
  readSensorsAvgTwo(pinoSensor1, pinoSensor2, v1_disp, v2_disp,
      vmed_disp);
  int h1_disp, h2_disp, h3_disp; float hmed_disp;
  readSensorsAvgThree(pinoSensorH1, pinoSensorH2, pinoSensorH3, h1_disp
      , h2_disp, h3_disp, hmed_disp);

  // escolher o texto de status
  char statusBuf[21];
  if (irrigando) {
    snprintf(statusBuf, sizeof(statusBuf), "Regando: Viveiro");
  } else if (irrigandoHorta) {
    snprintf(statusBuf, sizeof(statusBuf), "Regando: Horta");
  } else {
    snprintf(statusBuf, sizeof(statusBuf), "Solo OK");
  }

  char buf[21];
  lcd.setCursor(0, 0);
  // linha 0: m dia do viveiro
  snprintf(buf, sizeof(buf), "Viveiro:%3d%%", (int)vmed_disp);
  lcd.print(buf);
  lcd.setCursor(0, 1);
  // linha 1: m dia da horta
  snprintf(buf, sizeof(buf), "Horta:  %3d%%", (int)hmed_disp);
  lcd.print(buf);
  lcd.setCursor(0, 2);
```

```
29    // linha 2: data e hora curto
30    char dateBuf[21];
31    snprintf(dateBuf, sizeof(dateBuf), "%04d/%02d/%02d", now.year(), now.
          month(), now.day());
32    lcd.print(dateBuf);
33    lcd.setCursor(11, 2);
34    char timeBuf[10];
35    snprintf(timeBuf, sizeof(timeBuf), "%02d:%02d:%02d", now.hour(), now.
          minute(), now.second());
36    lcd.print(timeBuf);
37    lcd.setCursor(0, 3);
38    // linha 3: status (Regando Viveiro / Regando Horta / Solo OK)
39    lcd.print(statusBuf);
40
41    delay(100);
42  }
```

12: Exibição das informações no Display LCD

9.7 7. Atraso e Ciclo de Atualização

Por fim, o comando `delay(100)` insere uma pausa de 100 milissegundos entre os ciclos, limitando a taxa de atualização e reduzindo o consumo de energia do microcontrolador, sem comprometer a responsividade do sistema.