ANEXO C
FIRMWARE SSPA

Versión *1.0*

28/04/2024

INFORMACIÓN DEL PROYECTO

Autores			
Nombre completo	Capozzelli, Lucas Santiago	Camargo, Julián	Tedesco, Facundo
Legajo	42894	42741	42742
e-mail	lucascapozzelli@gmail.com	julicmrgo@gmail.com	Facu.Tedesco96@gmail.com

Tutor	Ing. Néstor Manzur
Director	Ing. Carlos Taffernaberry
Jurado	Ing. Carlos Taffernaberry
Año Académico	2024
Responsable de la cátedra	Ing. Antonio Álvarez Abril

Empresa / Cliente / Laboratorio	Instituto Nacional de Tecnología Agropecuaria (INTA)
Patrocinador (Sponsor)	Bodega Experimental del INTA

1- MÓDULO ADCSPA

- WeatherStation.hpp

Este archivo define la interfaz de la clase *WeatherStation*, que representa la estación meteorológica capaz de medir y recopilar datos climáticos. Esta interfaz incluye la declaración de los métodos y miembros de la clase, así como la especificación de sus funciones principales.

La clase *WeatherStation* contiene métodos para la inicialización de los sensores. Además, se declaran métodos para configurar y obtener los valores medidos por los sensores, como la velocidad del viento, la dirección del viento, la humedad, la radiación solar, la temperatura, la presión y la humedad de la hoja.

Entre las declaraciones, se especifican métodos como irrigateAndGetETc, que calcula y ejecuta el riego en base a parámetros específicos, y getPayload, que genera un string estructurado con la información recopilada por la estación meteorológica. Además, se incluyen métodos getter para obtener los valores medidos y otras variables internas de la estación meteorológica.

Este archivo sirve como la interfaz que guía la implementación en el archivo **WeatherStation.cpp**. Juntos, estos archivos constituyen la implementación de la lógica necesaria para adquirir, procesar y generar datos climáticos a través de la clase *WeatherStation*, siguiendo el paradigma de la programación orientada a objetos.

El código fuente se presenta a continuación:

```
#include <SFE_BMP180.h>
#include <TimeLib.h>
#include "HX711.h"
#include "DHT.h"

#define TIME_THRESHOLD 500
#define CALIBRATION 468.6
#define ATTEMPTS 5

#ifndef ESTACION_H
#define ESTACION_H

class WeatherStation {
    private:
        long initTime;
        long int windSpeed;
        int windDirection;
        float humidity;
        unsigned long int radiation;
        long int temperature;
        long int pressure;
        int leafMoisture;
        long pluviometerCounter;
        HX711 lysimeter;
        SFE_BMP180 bmp180Sensor;
```

```
        DHT *dht;

        /**
         * @brief cambio de escala entre floats
         */
        float fmap(float, float, float, float, float);

        /**
         * @brief funcion para volver a cero el contador de pulsos del
pluviometro
         */
        void resetPulseCounter();


    public:
        WeatherStation(long);

        WeatherStation();

// ------------------------------------ Inits ----------------------
-----

        /**
         * @brief Inicialización de los objetos bmp180Sensor y
lisímetro
         *
         */
        void init();

// ------------------------------------ Setters --------------------
-----

        /**
         * @brief Calcula la velocidad del viento.
         *
         * @param sensorVel del sensor
         * @return long int valor de velocidad del viento
         */
        void setWindSpeed(long int);

        /**
         * @brief Setea la dirección del viento
         *
         * @param sensorDir valor del sensor
         * @return int valor de la dirección del viento en grados
respecto      al norte
         */
        void setwindDirection(int);

        /**
         * @brief Setea la humedad del ambiente entre 0% y 100%
         *
         * @param sensorHum valor del sensor de SENSOR_HUMEDAD
         * @return uint humedad porcentaje de SENSOR_HUMEDAD
         */
```

```
        void setHumidity();

        /**
         * @brief Setea la radiación solar entre 0 y 1400 W/m2
         *
         * @param sensorRad valor leido del sensor de radiación solar
         * @return uint rad valor de radiacion solar
         */
        void setRadiation(long int);

        /**
         * @brief Lectura de la temperatura del sensor bmp180Sensor
         */
        void setTemperature();

        /**
         * @brief Lectura de la presion del sensor bmp180Sensor
         */
        void setPresion();

        /**
         * @brief Setea la humedad de hoja
         *
         * @param sensHoja valor leido del sensor
         * @return String result
         */
        void setLeafMoisture(int);

        /**
         * @brief funcion ISR que aumenta el numero de pulsos al
producirse
         * una interrupcion por flanco de subida en el pin.
         */
        void setPulseCounter (long int);

        /**
         * @brief Función que calcula el riego, y lo ejecuta
         *
         * @param
         * @return
         */
        float irrigateAndGetETc(float);
// ------------------------------------ Getters --------------------
-----

        /**
         * @brief Get the Vel Viento object
         *
         * @return long int
         */
        long int getWindSpeed();

        /**
         * @brief Get the Dir Viento object
         *
```

```
     * @return int
     */
    int getWindDirection();

    /**
     * @brief Get the Humedad object
     *
     * @return unsigned long int
     */
    float getHumidity();

    /**
     * @brief Get the Radiacion object
     *
     * @return unsigned long int
     */
    unsigned long int getRadiation();

    /**
     * @brief Get the Temperatura object
     *
     * @return long int
     */
    long int getTemperature();

    /**
     * @brief Get the Presion object
     *
     * @return long int
     */
    long int getPressure();

    /**
     * @brief Get the Hum Hoja object
     *
     * @return int with percentage of humidity in the leaf
     */
    int getLeafMoisture();

    /**
     * @brief Get the Contador Pluv object
     *
     * @return long
     */
    float getPluviometerCounter();

    /**
     * @brief Get the Peso Lisimetro object
     *
     * @return float
     */
    float getLysimeterWeight();

    /**
```

```
         * @brief Retorna el payload generado con los valores de la
estación
         *
         */
        String getPayload();

};
#endif
```

- WeatherStation.cpp

Este archivo implementa la funcionalidad definida en la clase *WeatherStation* del archivo de encabezado correspondiente **WeatherStation.hpp**. La clase representa la estación meteorológica que utiliza varios sensores para medir y recopilar datos climáticos.

El código comienza con la implementación de los constructores de la clase *WeatherStation*, inicializando variables como el tiempo de inicio y el contador del pluviómetro. Luego, se proporcionan métodos para la inicialización de los sensores, tales como el sensor BMP180 y el sensor DHT22. Además, se incluyen métodos para configurar los valores medidos por los sensores, como la velocidad del viento, la dirección del viento, la humedad, la radiación solar, la temperatura, la presión y la humedad de la hoja.

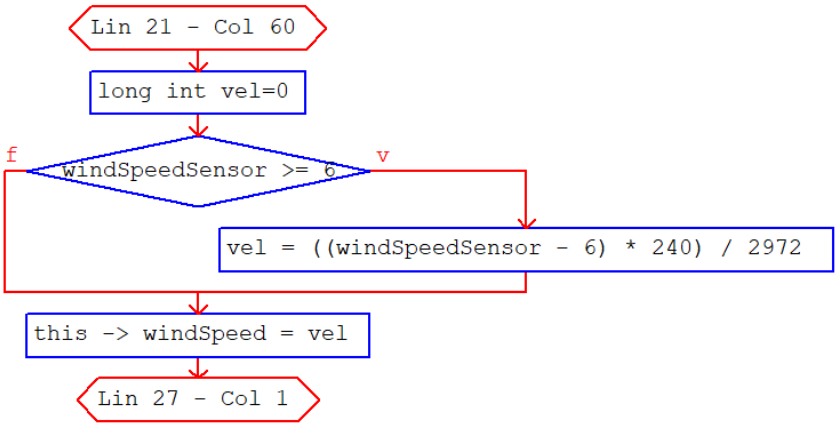
Entre los métodos implementados, se destaca la función irrigateAndGetETc, que calcula y ejecuta el riego en base a parámetros específicos, y la función getPayload, que genera un string estructurado con la información recopilada por la estación meteorológica para su posterior transmisión.

Adicionalmente, se proporcionan métodos getter para obtener los valores medidos y otras variables internas de la estación meteorológica.

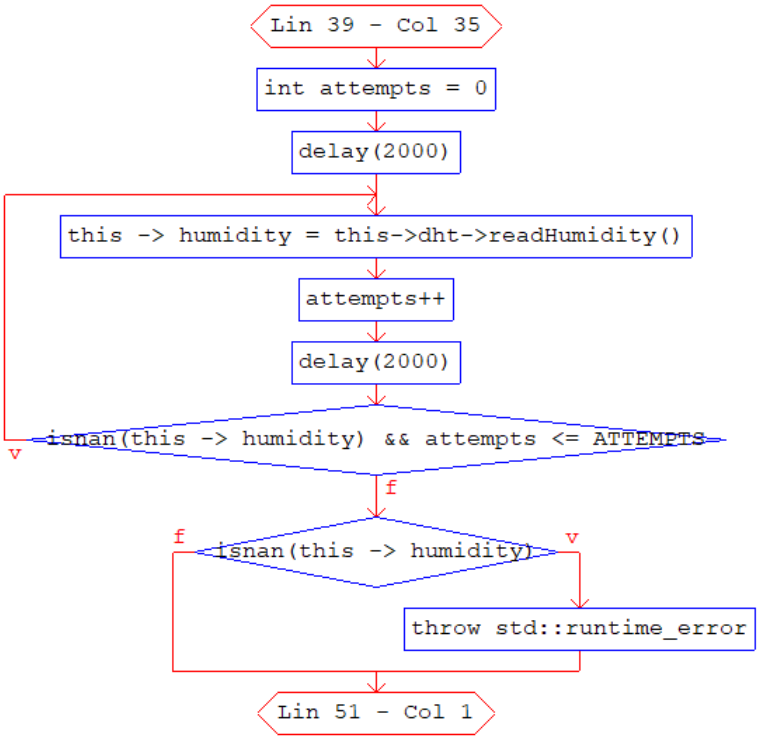
En resumen, el archivo contiene la implementación de la lógica necesaria para adquirir, procesar y generar datos climáticos a través de la clase *WeatherStation*. Los bloques de código encapsulados en los métodos operan en los datos de la instancia a la que pertenecen, siguiendo así el paradigma de la programación orientada a objetos.

A continuación, se presentan algunos diagramas en bloque de métodos pertenecientes a la clase *WeatherStation*, y el código fuente del programa:

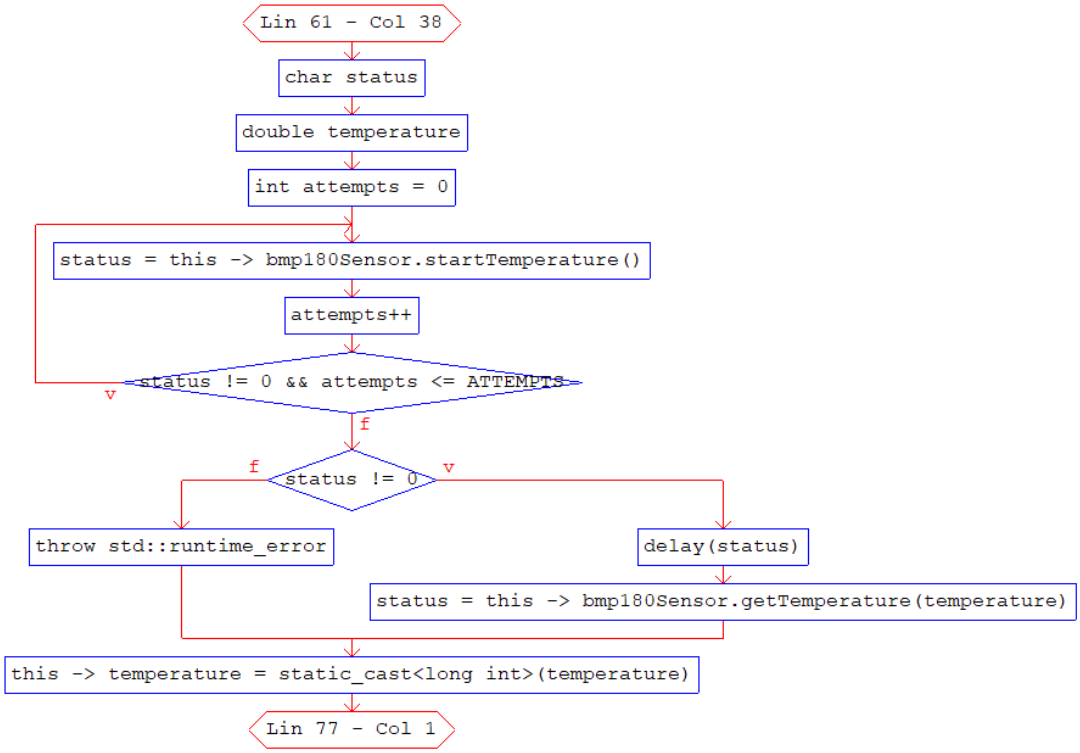
WeatherStation::setWindSpeed(long int windSpeedSensor)



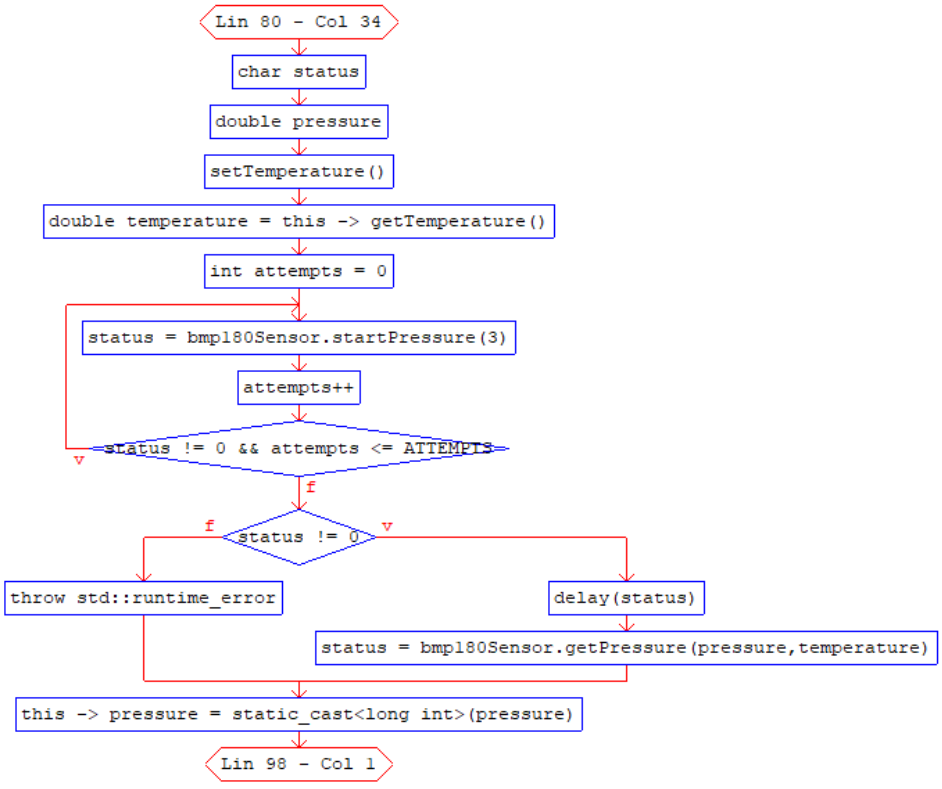
WeatherStation::setHumidity()



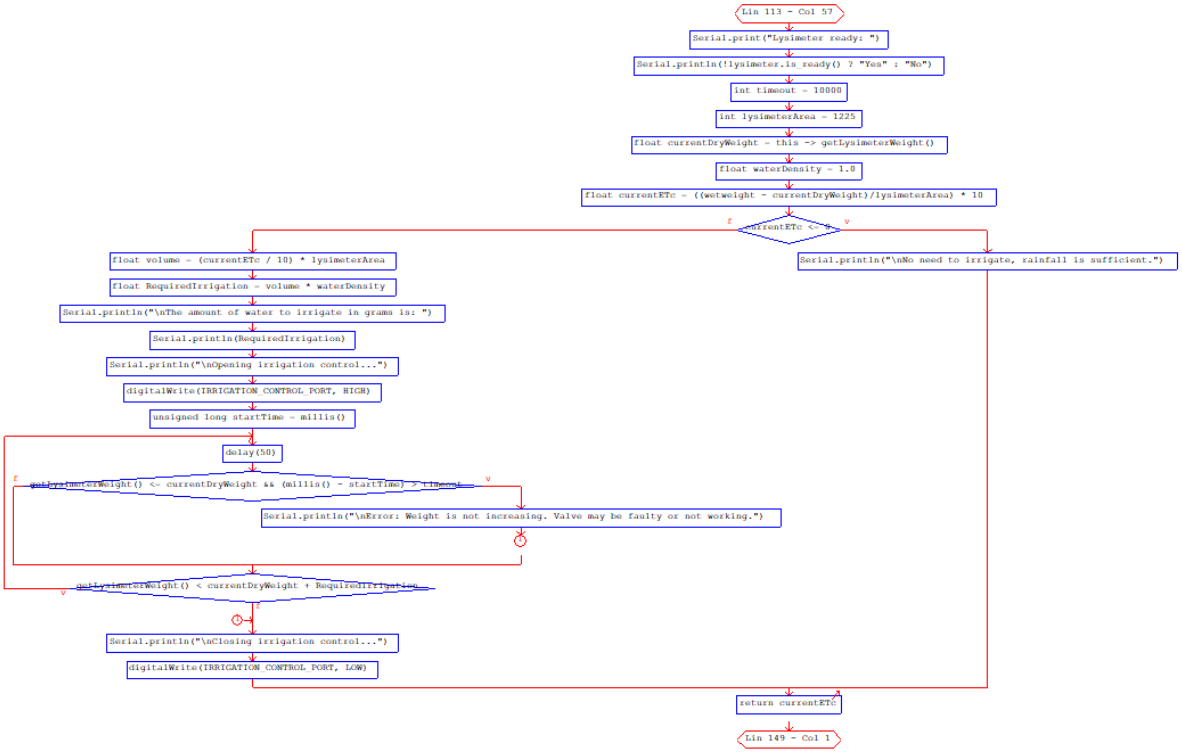
WeatherStation::setTemperature()



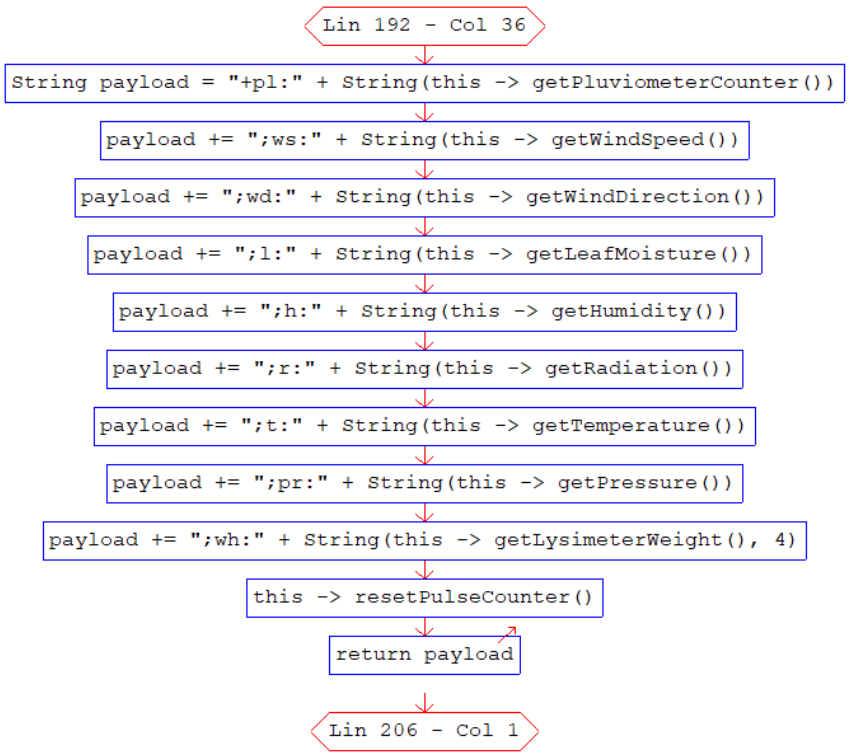
WeatherStation::setPresion()



WeatherStation:irrigateAndGetETc(float wetweight)



WeatherStation::getPayload()



A continuación, el código fuente:

```
#include "WeatherStation.hpp"

WeatherStation::WeatherStation(long initialCounter) {
    this -> initTime = 0;
    this -> pluviometerCounter = initialCounter;
}

WeatherStation::WeatherStation() {
    this -> initTime = 0;
    this -> pluviometerCounter = 0;
}
void WeatherStation::init() {
    this -> lysimeter.begin(16, 4);
    this -> lysimeter.set_scale(CALIBRATION);
    this -> lysimeter.tare();
    this -> bmp180Sensor.begin();
    this -> dht = new DHT(HUMIDITY_SENSOR_PORT, DHT22);
    this -> dht -> begin();
}

void WeatherStation::setWindSpeed(long int windSpeedSensor) {
    long int vel=0;
    if ( windSpeedSensor >= 6) {
        vel = ((windSpeedSensor - 6) * 240) / 2972;
    }
    this -> windSpeed = vel;
}

void WeatherStation::setwindDirection(int windDirectionSensor) {

    float valorVoltaje = fmap(windDirectionSensor, 0, 4095, 0.0, 3.3);
    this -> windDirection = (int) (valorVoltaje * 100);
}

float WeatherStation::fmap(float x, float in_min, float in_max, float
out_min, float out_max) {
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) +
out_min;
}

void WeatherStation::setHumidity() {
    int attempts = 0;
    delay(2000);
    do {
        this -> humidity = this->dht->readHumidity();
        attempts++;
        delay(2000);
    } while (isnan(this -> humidity) && attempts <= ATTEMPTS);

    if (isnan(this -> humidity)) {
        throw std::runtime_error("ERROR: Unable to obtain a valid
humidity reading after several attempts.");
    }
```

```
}

void WeatherStation::setRadiation(long int radiationSensor) {
    unsigned long int rad = 0;
    if (radiationSensor >= 270) {
        rad = ((radiationSensor - 270) * 1400) / 3651;
    }
    this -> radiation = rad > 1400 ? 1400 : rad;
}

void WeatherStation::setTemperature() {
    char status;
    double temperature;
    int attempts = 0;
    do {
        status = this -> bmp180Sensor.startTemperature();
        attempts++;
    } while(status != 0 && attempts <= ATTEMPTS);

    if (status != 0) {
        delay(status);
        status = this -> bmp180Sensor.getTemperature(temperature);
    } else {
        throw std::runtime_error("ERROR: Unable to obtain a valid
temperature reading.");
    }
    this -> temperature = static_cast<long int>(temperature);
}


void WeatherStation::setPresion() {
    char status;
    double pressure;
    setTemperature();
    double temperature = this -> getTemperature();
    int attempts = 0;
    do {
        status = bmp180Sensor.startPressure(3);
        attempts++;
    } while(status != 0 && attempts <= ATTEMPTS);

    if (status != 0){
        delay(status);
        status = bmp180Sensor.getPressure(pressure,temperature);
    } else {
        throw std::runtime_error("ERROR: Unable to obtain a valid
pressure reading.");
    }
    this -> pressure = static_cast<long int>(pressure);
}


void WeatherStation::setLeafMoisture(int leafHumididtySensor) {
    this -> leafMoisture = round((leafHumididtySensor*100)/2668);
}
```

```
void WeatherStation::setPulseCounter(long int currentCount) {
    this -> pluviometerCounter += currentCount;
}

void WeatherStation::resetPulseCounter() {
    this -> pluviometerCounter = 0;
}

float WeatherStation::irrigateAndGetETc(float wetweight) {
    // Check if the weight sensor is ready
    Serial.print("Lysimeter ready: ");
    Serial.println(!lysimeter.is_ready() ? "Yes" : "No");
    if (lysimeter.is_ready()) {
        throw std::runtime_error("ERROR:Unable to read the weight
sensor. Irrigation will not proceed.");
    } else {
        int timeout = 10000; // Timeout set to 10 seconds
        int lysimeterArea = 1225;  //cm2
        float currentDryWeight = this -> getLysimeterWeight();
        float waterDensity = 1.0; // Water density in g/cm³
        float currentETc = ((wetweight -
currentDryWeight)/lysimeterArea) * 10;
        if (currentETc <= 0) {
            Serial.println("\nNo need to irrigate, rainfall is
sufficient.");
        } else {
            float volume = (currentETc / 10) * lysimeterArea;
// Convert mm to cm³
            float RequiredIrrigation = volume * waterDensity;
// weight in grams
            Serial.println("\nThe amount of water to irrigate in grams
is: ");
            Serial.println(RequiredIrrigation);
// Verification of the weight to be added

            Serial.println("\nOpening irrigation control...");
            digitalWrite(IRRIGATION_CONTROL_PORT, HIGH);

            // Irrigate until the target weight is reached or timeout
            unsigned long startTime = millis();
            do {
                delay(50); // Wait 0.05 second between weight readings
                // Check if weight is increasing (indicating
successful irrigation)
                if (getLysimeterWeight() <= currentDryWeight &&
(millis() - startTime) > timeout) {
                    Serial.println("\nError: Weight is not increasing.
Valve may be faulty or not working.");
                    break;
                }
            } while (getLysimeterWeight() < currentDryWeight +
RequiredIrrigation);

            Serial.println("\nClosing irrigation control...");
```

```
            digitalWrite(IRRIGATION_CONTROL_PORT, LOW);
        }
        return currentETc;
    }
}


long int WeatherStation::getWindSpeed() {
    return this -> windSpeed;
}

int WeatherStation::getWindDirection() {
    return this -> windDirection;
}

float WeatherStation::getHumidity() {
    return this -> humidity;
}

unsigned long int WeatherStation::getRadiation() {
    return this -> radiation;
}

long int WeatherStation::getTemperature() {
    return this -> temperature;
}

long int WeatherStation::getPressure() {
    return this -> pressure;
}

int WeatherStation::getLeafMoisture() {
    return this -> leafMoisture;
}

float WeatherStation::getPluviometerCounter() {
    return this -> pluviometerCounter*0.25;
}

float WeatherStation::getLysimeterWeight() {
    return this -> lysimeter.get_units(4);
}


String WeatherStation::getPayload() {
    String payload = "+pl:" + String(this ->
getPluviometerCounter());       // pluviometer
    payload += ";ws:" + String(this ->  getWindSpeed());
// wind speed
    payload += ";wd:" + String(this ->  getWindDirection());
// wind direction
    payload += ";l:" + String(this ->  getLeafMoisture());
// leaf moisture
    payload += ";h:" + String(this ->  getHumidity());
// humidity
```

```
    payload += ";r:" + String(this ->  getRadiation());
// radiation
    payload += ";t:" + String(this ->  getTemperature());
// temperature
    payload += ";pr:" + String(this ->  getPressure());
// pressure
    payload += ";wh:" + String(this ->  getLysimeterWeight(), 4);
// weight

    this -> resetPulseCounter();

    return payload;
}
```

- main.cpp

Este archivo constituye el punto de entrada principal del firmware. En este código, se implementa la lógica central que coordina la comunicación, la adquisición de datos climáticos y la interacción con todos los sensores representados por la clase *WeatherStation*. A continuación, se describen los elementos clave del archivo:

El código inicia declarando e incluyendo las bibliotecas necesarias, así como las instancias de las clases *WeatherStation*, *ATFunctions*, y *HexFunctions*.

En la función setup**()**, se realiza la configuración inicial del entorno, que incluye la inicialización de la comunicación serial, la configuración de pines, la conexión a los sensores y la configuración del módulo LoRa RAK3172 mediante comandos AT. También se establecen interrupciones ligadas al registro de precipitaciones debido al funcionamiento del pluviómetro.

La función loop() encapsula la lógica de control central del sistema, coordinando la adquisición de datos meteorológicos, la gestión de comandos, y la transmisión de información a través del módulo LoRa RAK3172. La función inicia comprobando si hay datos disponibles en el puerto serial. En caso afirmativo, lee el comando recibido por el modulo LoRa. Si el comando recibido comienza con los identificadores POLL_COMMAND o IRR_COMMAND, se realiza una serie de acciones para actualizar los valores del lisímetro de pesada y la estación meteorológica representada por la clase *WeatherStation*. Se obtienen lecturas de sensores, como la velocidad y dirección del viento, la humedad, la radiación solar, la temperatura, la presión, la humedad de la hoja y las precipitaciones, y se almacenan en la instancia de WeatherStation. Se genera un paquete de transmisión con la información recopilada utilizando el método getPayload() de la instancia de WeatherStation.

Si el comando es de riego (IRR_COMMAND), se procesa la información adicional contenida en el comando para determinar la cantidad de riego necesario. Se realiza el riego mediante el método irrigateAndGetETc(), y se ajusta el paquete de transmisión con información adicional relacionada con el riego. Se prepara el paquete de transmisión con marcas de inicio y fin (FRAME_START y FRAME_END). Se muestra el paquete antes de enviarlo, y se ejecutan comandos AT para configurar la transmisión con el módulo LoRa. Se envía el paquete mediante comandos AT y se imprime la respuesta recibida.

A continuación, el código fuente:

```
#include <stdio.h>
#include <Arduino.h>
#include <TimeLib.h>
#include <Utils.hpp>
#include <HardwareSerial.h>
#include "WeatherStation.hpp"

#define FRAME_START ">"
#define FRAME_END "<"

volatile static long int pluvCounter = 0;
long startTime = 0;
long int initialTime = 0;

void pulseDetector();
WeatherStation weatherStation;
ATFunctions atFunctions;
HexFunctions hexFunctions;

void setup() {
  Serial.begin(115200);
  Serial1.begin(115200, SERIAL_8N1, 19, 5); //RX, TX

  Serial.println(atFunctions.sendATCommand(Serial1, AT_RESET));
  Serial.println(atFunctions.sendATCommand(Serial1, AT_SET_P2P_MODE));
  Serial.println(atFunctions.sendATCommand(Serial1,
AT_BAUD_115200_CONFIG_SET));
  Serial.println(atFunctions.sendATCommand(Serial1,
AT_P2P_CONFIG_SET));
  Serial.println(atFunctions.sendATCommand(Serial1,
AT_P2P_CONFIG_GET));
  Serial.println(atFunctions.sendATCommand(Serial1,
AT_CONTINUOUS_PRECV_CONFIG_SET));

  weatherStation.init();

  pinMode(WIND_SPEED_SENSOR_PORT, INPUT);
  pinMode(WIND_DIRECTION_SENSOR_PORT, INPUT);
  pinMode(RADIATION_SENSOR_PORT, INPUT);
  pinMode(HUMIDITY_SENSOR_PORT, INPUT);
  pinMode(TEMPERATURE_SENSOR_PORT, INPUT);
  pinMode(LEAF_MOISTURE_SENSOR_PORT, INPUT);
  pinMode(IRRIGATION_CONTROL_PORT, OUTPUT);

  digitalWrite(IRRIGATION_CONTROL_PORT, LOW);

  attachInterrupt(digitalPinToInterrupt(PLUVIOMETRO_PORT),
pulseDetector, RISING);

  Serial.println("Setup finished");
}

void loop() {
```

```
  if(Serial1.available()>0) {
    String rxData = atFunctions.readSerial(Serial1);
    rxData.trim();
    rxData =
hexFunctions.hexToASCII(rxData.substring(rxData.lastIndexOf(':')+1));
    Serial.print("Instruction received: ");
    Serial.println(rxData);
  try {
    if (rxData.startsWith(POLL_COMMAND) ||
rxData.startsWith(IRR_COMMAND)) {

weatherStation.setWindSpeed(analogRead(WIND_SPEED_SENSOR_PORT));

weatherStation.setwindDirection(analogRead(WIND_DIRECTION_SENSOR_PORT)
);
        weatherStation.setHumidity();

weatherStation.setRadiation(analogRead(RADIATION_SENSOR_PORT));
        weatherStation.setTemperature();
        weatherStation.setPresion();

weatherStation.setLeafMoisture(analogRead(LEAF_MOISTURE_SENSOR_PORT));
        weatherStation.setPulseCounter(pluvCounter);

        String transmitionPacket = weatherStation.getPayload();

        if (rxData.startsWith(IRR_COMMAND)) {

          String commmandData =
rxData.substring(rxData.indexOf(';')+1);
          float wetweight = commmandData.substring(0,
commmandData.indexOf(';')+1).toFloat();
          float ETc = weatherStation.irrigateAndGetETc(wetweight);
//controla el riego con wetweight y la lluvia consultada
          transmitionPacket = FRAME_START + String(IRR_COMMAND) +
transmitionPacket + FRAME_END;
          transmitionPacket = transmitionPacket.substring(0,
transmitionPacket.length()-1);
          transmitionPacket += ";etc:" + String(ETc, 2);
          transmitionPacket += ";wwh:" +
String(weatherStation.getLysimeterWeight()) + FRAME_END;
          Serial.println(transmitionPacket);
        } else {
          transmitionPacket = FRAME_START + String(POLL_COMMAND) +
transmitionPacket + FRAME_END;
        }
        Serial.print("Sending packet:");
        Serial.println(transmitionPacket);
        atFunctions.sendATCommand(Serial1, AT_P2P_CONFIG_TX_SET);
        String response = atFunctions.sendP2PPacket(Serial1,
transmitionPacket);
        Serial.print("Response: ");
        response.replace('\n', ' ');
        Serial.println(response);
```

```
        atFunctions.sendATCommand(Serial1,
AT_CONTINUOUS_PRECV_CONFIG_SET);

        pluvCounter = 0;
        startTime = millis();
        delay(300);
      }
    } catch (std::runtime_error& e) {
      Serial.println(e.what());
      atFunctions.sendATCommand(Serial1, AT_P2P_CONFIG_TX_SET);
      atFunctions.sendP2PPacket(Serial1, e.what());
      atFunctions.sendATCommand(Serial1,
AT_CONTINUOUS_PRECV_CONFIG_SET);
    }
  }
}

void pulseDetector() {
    if(millis() - initialTime > TIME_THRESHOLD){
        pluvCounter++;
        initialTime = millis();
    }
}
```

2- MÓDULO RXSPA

- Logger.hpp

Este archivo define la implementación de una clase llamada *Logger* que se utiliza para realizar el registro (logging) de mensajes en el servidor a través de HTTP (es el archivo de encabezado de la clase). La clase tiene un método llamado "config" que se utiliza para configurar el logger. También tiene otro método llamado "log" que se utiliza para enviar mensajes de registro al servidor. Además del método "log", se proporcionan métodos específicos para distintos niveles de registro (error, info, y debug). Cada método de nivel de registro toma los mismos parámetros que el método log, pero el nivel correspondiente ya está predefinido.

La clase contiene miembros privados para almacenar la información de conexión como el host, el nombre de usuario, la contraseña, y un objeto *HTTPClient* para realizar las solicitudes HTTP.

El código fuente se presenta a continuación:

```
#include <Arduino.h>
#include <HTTPClient.h>
#include <thread>

#define ERROR_LEVEL "ERROR"
#define INFO_LEVEL "INFORMATION"
#define DEBUG_LEVEL "DEBUG"

#define RXSPA "RXSPA"

#ifndef LOGGER_H
#define LOGGER_H

class Logger {
    public:

        /**
         * @brief Configure the logger
         *
         * @param logHost The host to send the logs
         * @param user The user to authenticate on the host
         * @param pass The password to authenticate on the host
        */
        void config(String logHost, String user, String pass, int
attempts = 3);

        /**
         * @brief Log a message to the backend
         *
         * @param httpcode The http code of the request
         * @param message The message to log
         * @param level The level of the message
```

```
         * @return int The http code of the request
        */
        int log(int httpcode, String message, String level, String
source);

        /**
         * @brief Log a message to the backend in a new thread
(parallel)
        */

        int error(int httpCode, String message, String source =
RXSPA);
        int info(int httpCode, String message, String source = RXSPA);
        int debug(int httpCode, String message, String source =
RXSPA);

    private:
        String logHost;
        String user;
        String pass;
        HTTPClient http;
        int attempts;
};
#endif
```

- Logger.cpp

En este archivo, se proporciona la implementación (definición) detallada de cada método de la clase *Logger* que fue declarada en Logger.hpp. Incluye el archivo de encabezado "Logger.hpp" para que el compilador tenga acceso a la declaración de la clase y sus miembros.

El método “config” asigna los valores de los parámetros proporcionados a los miembros correspondientes de la clase (logHost, user, pass, y attempts).

El método “log” construye un mensaje de registro en formato JSON que incluye información como el código HTTP, el mensaje, el nivel y la fuente del mensaje. Utiliza un bucle para intentar enviar el mensaje al servidor. Dentro del bucle, se configuran los encabezados HTTP, la autorización con el nombre de usuario y la contraseña, y se realiza la solicitud POST al servidor. Si la respuesta del servidor no es exitosa (código HTTP diferente de 201), se muestra un mensaje de error y se intenta nuevamente, hasta que se alcance el número máximo de intentos o se obtenga una respuesta exitosa. El código de respuesta HTTP final se devuelve.

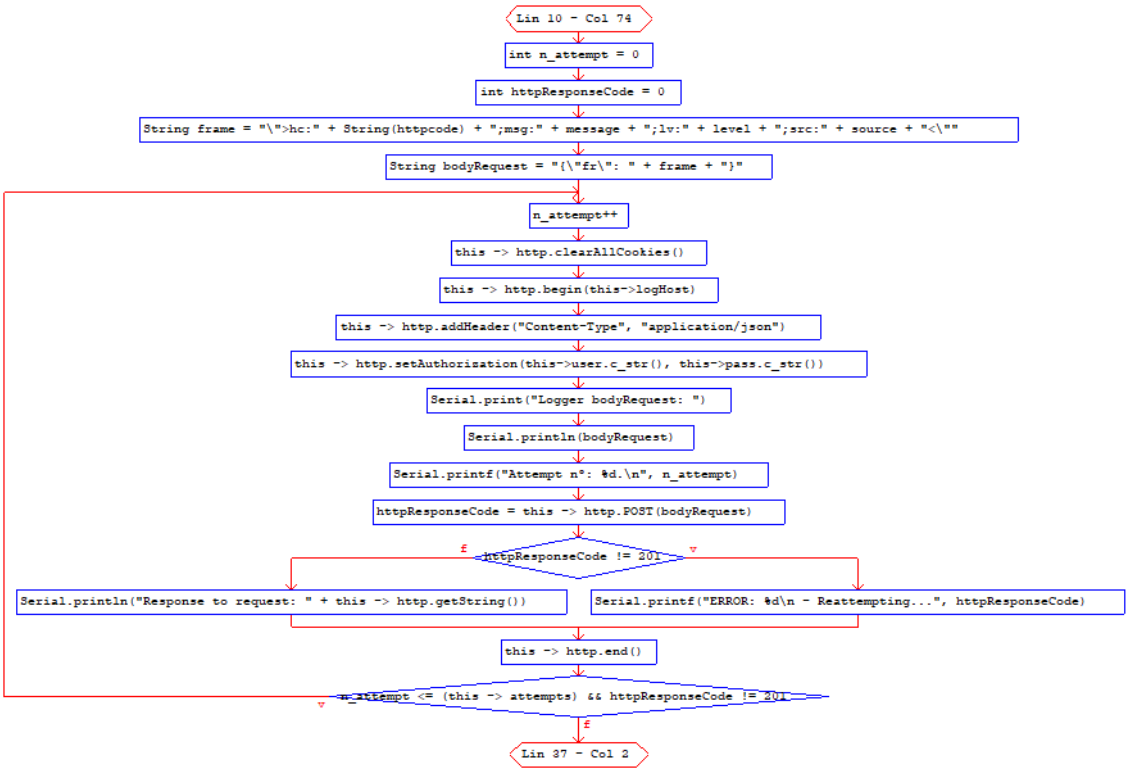
Los métodos de niveles de registro (error, info, debug), simplemente llaman al método” log” con el nivel correspondiente predefinido (ERROR_LEVEL, INFO_LEVEL, DEBUG_LEVEL).

En resumen, el archivo Logger.cpp implementa las funciones necesarias para realizar el registro de mensajes en el servidor utilizando HTTP. Configura la conexión HTTP con la

información proporcionada en el método “config” y proporciona funciones específicas para diferentes niveles de registro (error, info, debug). El método “log” se encarga del envío real del mensaje de registro y maneja los intentos en caso de fallos.

A continuación, se presenta el diagrama en bloque del método “log” de la clase *Logger*:

Logger::log(int httpcode, String message, String level, String source)



Ahora, el código fuente del archivo Logger.cpp:

```
#include "Logger.hpp"

void Logger::config(String logHost, String user, String pass, int
attempts) {
    this -> logHost = logHost;
    this -> user = user;
    this -> pass = pass;
    this -> attempts = attempts;
}

int Logger::log(int httpcode, String message, String level, String
source){
    int n_attempt = 0;
    int httpResponseCode = 0;
    String frame =  "\">hc:" + String(httpcode) +
                    ";msg:" + message +
                    ";lv:" + level +
                    ";src:" + source + "<\"";
    String bodyRequest = "{\"fr\": " + frame + "}";
    try {
```

```
        do {
            n_attempt++;
            this -> http.clearAllCookies();
            this -> http.begin(this->logHost);
            this -> http.addHeader("Content-Type",
"application/json");
            this -> http.setAuthorization(this->user.c_str(), this-
>pass.c_str());
            Serial.print("Logger bodyRequest: ");
            Serial.println(bodyRequest);
            Serial.printf("Attempt n°: %d.\n", n_attempt);
            httpResponseCode = this -> http.POST(bodyRequest);
            if(httpResponseCode != 201) {
                Serial.printf("ERROR: %d\n - Reattempting...",
httpResponseCode);
            } else {
                Serial.println("Response to request: " + this ->
http.getString());
            }
            this -> http.end();
        } while(n_attempt <= (this -> attempts) && httpResponseCode !=
201);
    } catch (const std::exception& e) {
        httpResponseCode = -1;
        Serial.println(e.what());
        this -> http.end();
    }
    return httpResponseCode;
}

int Logger::error(int httpcode, String message, String source) {
    return this -> log(httpcode, message, ERROR_LEVEL, source);
}

int Logger::info(int httpcode, String message, String source) {
    return this -> log(httpcode, message, INFO_LEVEL, source);
}

int Logger::debug(int httpcode, String message, String source) {
    return this -> log(httpcode, message, DEBUG_LEVEL, source);
}
```

- RestCall.hpp

Este archivo define la estructura de la clase *RestCall*, que se encarga de facilitar las operaciones de llamadas REST hacia el servidor. La clase proporciona métodos para configurar la conexión, enviar comandos específicos al servidor, y obtener respuestas. Para su funcionamiento, la clase utiliza la biblioteca HTTPClient.h para realizar solicitudes HTTP y ArduinoJson.h para manipulación de datos en formato JSON. Además, se incluye la clase *Logger* para gestionar los registros y niveles de depuración.

REST (Representational State Transfer) es un enfoque para diseñar servicios web de manera sencilla y eficiente. Se basa en principios como el uso de recursos identificables por URLs y la realización de operaciones estándar (GET, POST, PUT, DELETE) sobre estos recursos.

En la clase *RestCall*, el uso de REST implica enviar solicitudes HTTP para realizar acciones específicas, como obtener información (getWeight), enviar datos (sendFrameData), o verificar la disponibilidad del servidor (ping). Esto se hace de manera estructurada y sigue principios que promueven la simplicidad y la independencia entre el cliente y el servidor.

El método "config" se encarga de configurar la instancia de la clase con la URL del host, el nombre de usuario, y la contraseña necesarios para autenticarse en el servidor. Esto establece la base para las operaciones subsiguientes.

El método "getWeight" realiza una solicitud al servidor con un comando específico y devuelve la respuesta correspondiente. Esta función está diseñada para obtener valores de peso desde el backend.

El método "sendFrameData" permite enviar datos de un marco al servidor, especificando la tabla de destino y el número de intentos en caso de falla en el envío.

El método "ping" envía un ping al servidor para verificar su disponibilidad. El número de intentos permite controlar la tolerancia a fallos en la conexión.

Adicionalmente, la clase proporciona métodos para configurar y obtener el nivel de depuración, así como establecer y recuperar el código de respuesta de la última solicitud HTTP realizada.

El código fuente se presenta a continuación:

```
#include <Arduino.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <Logger.hpp>

#define STATION_TABLE "spa.weatherstation"
#define WET_WEIGHT_TABLE "spa.wetweights"
#define ETC_TABLE "spa.etc"

#define INSERT_CONTEXT "/insert"
#define LOG_CONTEXT "/log"
#define ETCRAIN_CONTEXT "/etc"
#define PING_CONTEXT "/ping"

#define ERROR_LEVEL "ERROR"
#define INFO_LEVEL "INFORMATION"
#define DEBUG_LEVEL "DEBUG"

class RestCall {
    public:

        /**
```

```
         * @brief Configure the rest call
         *
         * @param apiUrl The host to send the data
         * @param dbUser The user to authenticate on the host
         * @param dbPass The password to authenticate on the host
        */
        void config(String apiUrl, String dbUser, String dbPass);

        /**
         * @brief Get the weight and rain values from the backend
         *
         * @param command The command to send to the backend
         * @return String The response of the backend
        */
        String getWeight(String command);

        /**
         * @brief Send a frame to the backend
         *
         * @param frame The frame to send
         * @param table The table to insert the frame
         * @param attempts The number of attempts to send the frame
         * @return String The response of the backend
        */
        String sendFrameData(String frame, String table, int
attempts);

        /**
         * @brief Send a ping to the backend for up the server
         *
        */
        String ping(int attempts);

        /**
         * @brief Set the debug level
         *
         * @param debugLevel The debug level to set
         * @return String The debug level
        */
        void setDebugLevel(String debugLevel);

        /**
         * @brief Get the debug level
         *
         * @return String The debug level
        */
        String getDebugLevel();

        /**
         * @brief Set the response code
         *
         * @param responseCode The response code to set
         * @return int The response code
        */
        void setResponseCode(int responseCode);
```

```
    /**
     * @brief Get the response code
     *
     * @return int The response code
    */
    int getResponseCode();

private:
    String apiUrl;
    String dbUser;
    String dbPass;
    HTTPClient http;
    String debugLevel;
    int responseCode;
};
```

- RestCall.cpp

Este archivo contiene la implementación real de las funciones declaradas en RestCall.hpp. Aquí es donde se detallan las acciones específicas de cada método, se maneja la lógica interna, y se utilizan las bibliotecas y herramientas necesarias para realizar las operaciones deseadas.

La clase *RestCall* encapsula la lógica para interactuar con el servidor mediante REST, ofreciendo una interfaz simple para realizar operaciones como obtención de datos, envío de información y verificación del estado del servidor. La integración con la clase *Logger* permite mantener un registro estructurado de las operaciones y facilita el seguimiento de posibles problemas en la comunicación con el servidor.

El método config asigna las variables proporcionadas a los miembros correspondientes de la clase, estableciendo así la información necesaria para las solicitudes HTTP.

Los métodos setDebugLevel y getDebugLevel permiten configurar y obtener el nivel de depuración de la clase, respectivamente.

Los métodos setResponseCode y getResponseCode se utilizan para establecer y obtener el código de respuesta de la última solicitud HTTP realizada.

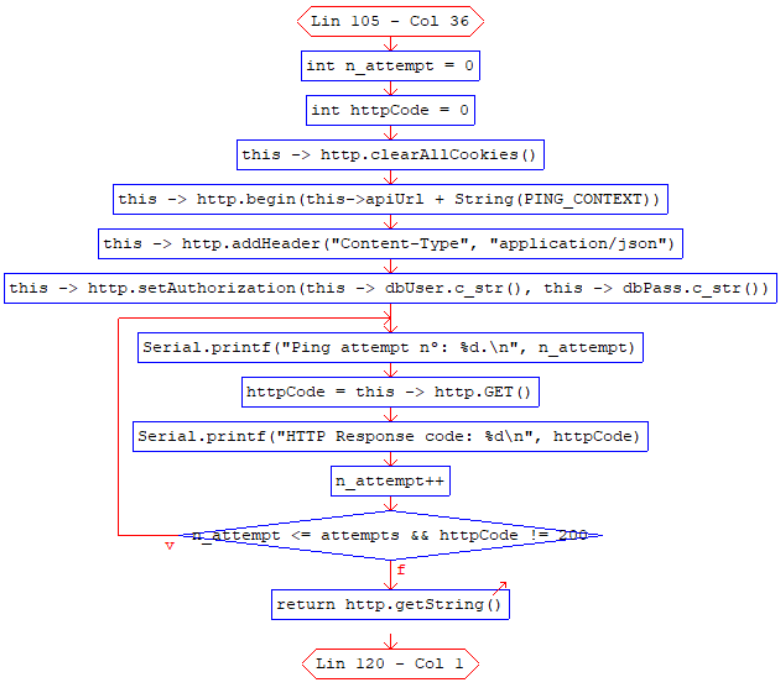
El método sendFrameData envía un marco de datos al servidor, permitiendo reintentos en caso de respuestas no exitosas. Gestiona el registro de eventos y establece el código de respuesta y nivel de depuración correspondientes.

El método getWeight realiza una solicitud al servidor para obtener valores de peso, con reintentos en caso de respuestas no exitosas. Analiza la respuesta JSON del servidor para extraer el valor correspondiente.

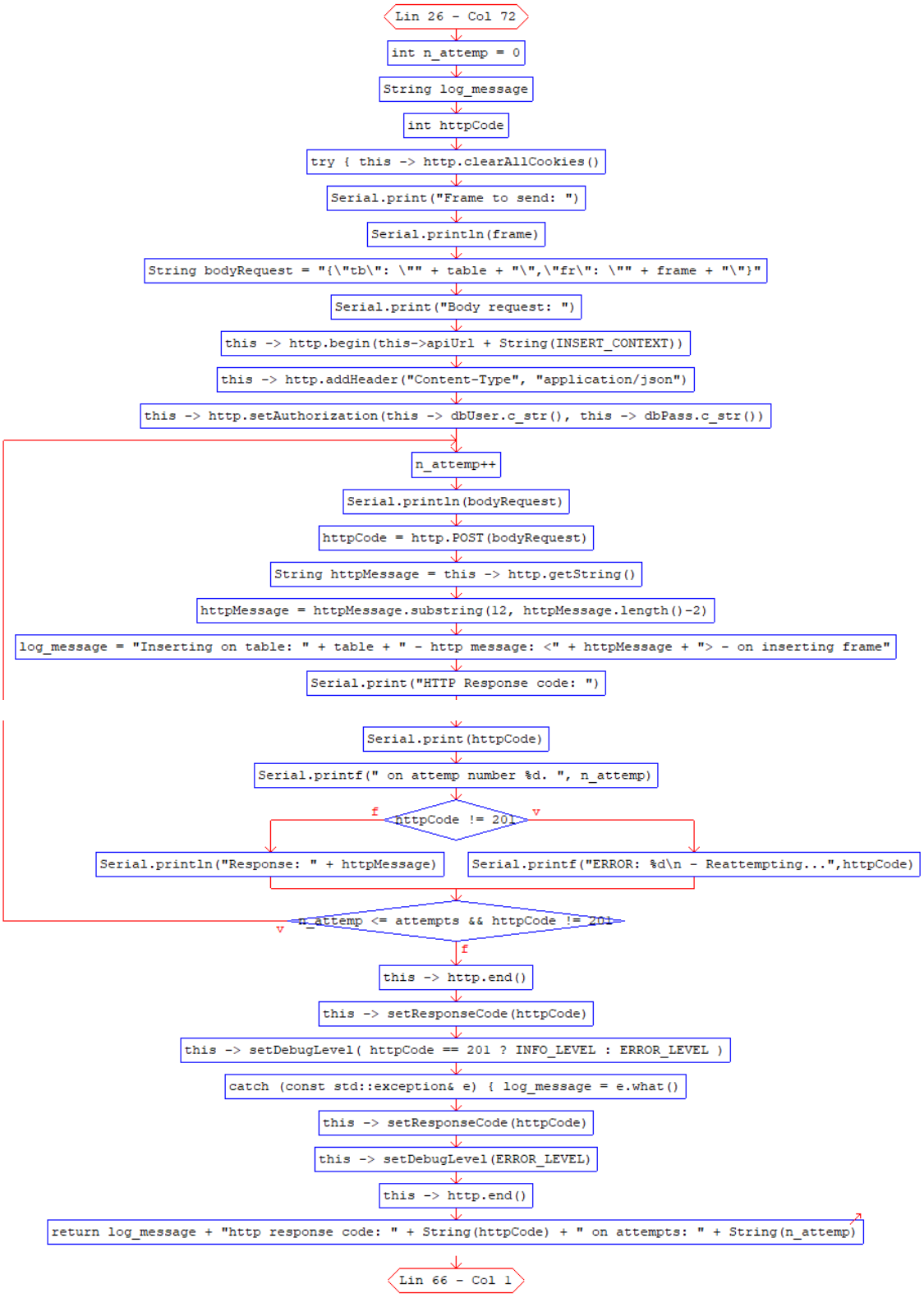
El método ping envía un ping al servidor para verificar su disponibilidad, con reintentos en caso de respuestas no exitosas.

A continuación, se presentan los diagramas en bloque de algunos métodos de la clase *RestCall*:

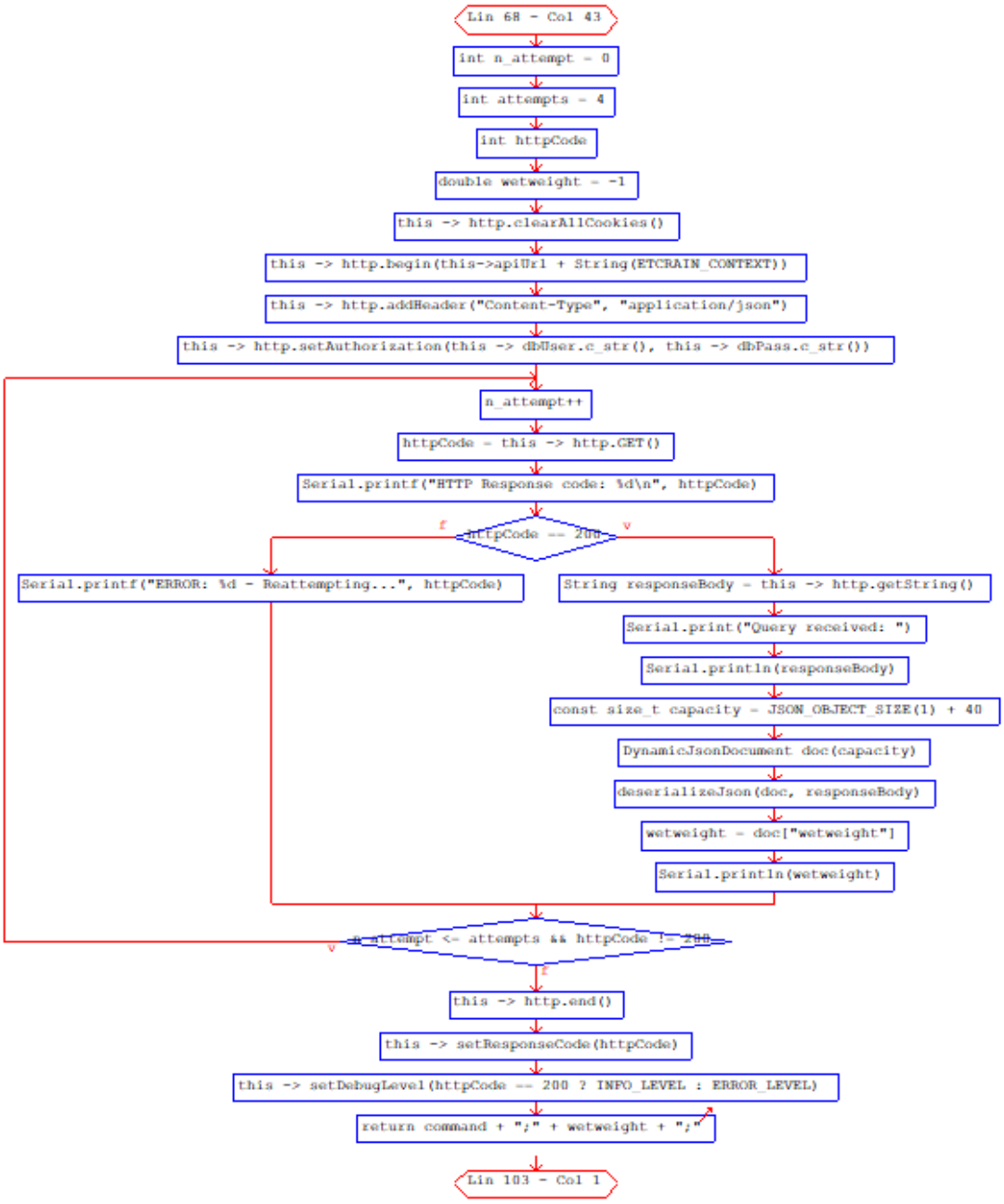
RestCall::ping(int attempts)



String RestCall::sendFrameData(String frame, String table, int attempts)



RestCall::getWeight(String command)



Ahora, el código fuente del archivo RestCall.cpp:

```
#include "RestCall.hpp"

void RestCall::config(String apiUrl, String dbUser, String dbPass) {
    this -> apiUrl = apiUrl;
    this -> dbUser = dbUser;
    this -> dbPass = dbPass;
    this -> http.setTimeout(60000);
}
```

```
void RestCall::setDebugLevel(String debugLevel) {
    this -> debugLevel = debugLevel;
}

String RestCall::getDebugLevel() {
    return this -> debugLevel;
}

void RestCall::setResponseCode(int responseCode) {
    this -> responseCode = responseCode;
}

int RestCall::getResponseCode() {
    return this -> responseCode;
}

String RestCall::sendFrameData(String frame, String table, int
attempts){
    int n_attemp = 0;
    String log_message;
    int httpCode;
    try {
        this -> http.clearAllCookies();
        Serial.print("Frame to send: ");
        Serial.println(frame);
        String bodyRequest = "{\"tb\": \"" + table + "\",\"fr\": \"" +
frame + "\"}";
        Serial.print("Body request: ");
        this -> http.begin(this->apiUrl + String(INSERT_CONTEXT));
        this -> http.addHeader("Content-Type", "application/json");
        this -> http.setAuthorization(this -> dbUser.c_str(), this ->
dbPass.c_str());
        do {
            n_attemp++;
            Serial.println(bodyRequest);
            httpCode = http.POST(bodyRequest);
            String httpMessage = this -> http.getString();
            httpMessage = httpMessage.substring(12,
httpMessage.length()-2);
            log_message = "Inserting on table: " + table + " - http
message: <" + httpMessage + "> - on inserting frame";
            Serial.print("HTTP Response code: ");
            Serial.print(httpCode);
            Serial.printf(" on attemp number %d. ", n_attemp);
            if(httpCode != 201) {
                Serial.printf("ERROR: %d\n -
Reattempting...",httpCode);
            } else {
                Serial.println("Response: " + httpMessage);
            }
        } while (n_attemp <= attempts && httpCode != 201);
        this -> http.end();
        this -> setResponseCode(httpCode);
```

```
        this -> setDebugLevel( httpCode == 201 ? INFO_LEVEL :
ERROR_LEVEL );
    } catch (const std::exception& e) {
        log_message = e.what();
        this -> setResponseCode(httpCode);
        this -> setDebugLevel(ERROR_LEVEL);
        this -> http.end();
    }

    return log_message + "http response code: " + String(httpCode) + "
on attempts: " + String(n_attemp);
}

String RestCall::getWeight(String command) {
    int n_attempt = 0;
    int attempts = 4;
    int httpCode;
    double wetweight = -1;

    this -> http.clearAllCookies();
    this -> http.begin(this->apiUrl + String(ETCRAIN_CONTEXT));
    this -> http.addHeader("Content-Type", "application/json");
    this -> http.setAuthorization(this -> dbUser.c_str(), this ->
dbPass.c_str());

    do {
        n_attempt++;
        httpCode = this -> http.GET();
        Serial.printf("HTTP Response code: %d\n", httpCode);

        if (httpCode == 200) {
            String responseBody = this -> http.getString();
            Serial.print("Query received: ");
            Serial.println(responseBody);
            const size_t capacity = JSON_OBJECT_SIZE(1) + 40;
            DynamicJsonDocument doc(capacity);
            deserializeJson(doc, responseBody);
            wetweight = doc["wetweight"];
            Serial.println(wetweight);
        } else {
            Serial.printf("ERROR: %d - Reattempting...", httpCode);
        }
    } while (n_attempt <= attempts && httpCode != 200);

    this -> http.end();
    this -> setResponseCode(httpCode);
    this -> setDebugLevel(httpCode == 200 ? INFO_LEVEL : ERROR_LEVEL);

    return command + ";" + wetweight + ";";
}

String RestCall::ping(int attempts) {
    int n_attempt = 0;
    int httpCode = 0;
    this -> http.clearAllCookies();
```

```
    this -> http.begin(this->apiUrl + String(PING_CONTEXT));
    this -> http.addHeader("Content-Type", "application/json");
    this -> http.setAuthorization(this -> dbUser.c_str(), this ->
dbPass.c_str());
    do {
        Serial.printf("Ping attempt n°: %d.\n", n_attempt);
        httpCode = this -> http.GET();
        Serial.printf("HTTP Response code: %d\n", httpCode);
        n_attempt++;
    } while(n_attempt <= attempts && httpCode != 200);

    return http.getString();
}
```

- main.cpp

Este archivo es el punto de entrada principal del programa y contiene la lógica principal de la aplicación. Orquesta la ejecución del programa, gestionando la configuración inicial, las llamadas a funciones específicas en momentos predeterminados y las acciones periódicas para mantener la comunicación con el servidor y realizar operaciones programadas.

Inicia la ejecución definiendo la configuración inicial del programa, que incluye la inicialización de la comunicación serial y la conexión WiFi a través de WiFiManager. Además, establece instancias, como las clases *Logger* y *RestCall*, para gestionar registros y realizar llamadas a la API mediante REST. Dentro de su estructura, se definen constantes que representan URLs, comandos específicos y otras configuraciones relevantes. Se inician instancias de clases útiles, como *Timestamp*, que proporciona información sobre la fecha y hora actuales. Organiza la ejecución del programa mediante la configuración de alarmas. Se definen dos alarmas, pollAlarm e irrAlarm, que invocan la función sendPollCommand con comandos específicos.

La función sendPollCommand gestiona la comunicación entre el módulo RXSPA y el modulo ADCSPA, como asi también la comunicación con el servidor del Sistema SPA. La función comienza enviando un comando al módulo ADCSPA mediante el uso del metódo sendP2PPacket(Serial2, pollCommand) de la clase atFunctions. Este comando específico se espera que genere una respuesta por parte del módulo ADCSPA, y la función configura el módulo LoRa para recibir dicha respuesta. Posteriormente, la función entra en un bucle de espera, donde monitorea la llegada de datos al puerto serie. Si se recibe una respuesta, se interpreta y se almacena en la variable frame. Además, se registra este marco de datos mediante la instancia de la clase *Logger* para llevar un registro detallado de la operación. La función maneja la lógica de reintentos en caso de no recibir una respuesta exitosa dentro del tiempo especificado. Si no se recibe un marco de datos o si se supera el número máximo de intentos, la función realiza una serie de acciones, como registrar un error, reconfigurar la comunicación serial y volver a enviar el comando.

Si se recibe un marco de datos sin errores, la función utiliza la instancia de la clase *RestCall* para enviar este marco al servidor mediante restCall.sendFrameData(frame, STATION_TABLE, 3). La respuesta de esta operación se registra nuevamente utilizando la instancia de la clase *Logger*. En el caso específico de comandos relacionados con la irrigación (IRR_COMMAND), la función realiza una operación adicional. Divide el marco de datos en tres partes correspondientes a diferentes tablas en la base de datos (STATION_TABLE, ETC_TABLE, y WET_WEIGHT_TABLE). Cada parte se envía por separado al servidor mediante llamadas a restCall.sendFrameData, y las respuestas se registran con la instancia de Logger.

En el bucle principal, se gestiona la lógica continua del programa. Se obtiene la fecha y hora actuales, y se verifica si es necesario ejecutar las funciones asociadas a las alarmas. Además, se realiza un ping al servidor y se imprime la marca de tiempo local cuando los minutos son igual a 55, proporcionando una comprobación periódica del estado del sistema.

A continuación, el código fuente del archivo main.cpp:

```
#include <Arduino.h>
#include <WiFiManager.h>
#include <Utils.hpp>
#include <esp_sntp.h>
#include <time.h>
#include <Logger.hpp>
#include <RestCall.hpp>
#include "secrets.h"

#define POOL_NTP_URL "pool.ntp.org"

#define API_URL "https://backend-spa.onrender.com"

#define IRR_HOUR "03"
#define POLL_MINUTES "00"

#define ETC "etc"
#define WET_WEIGHT "wwh"
#define IRR_PREFIX ">IRR+"
#define FRAME_CLOSE "<"
#define ERROR "ERROR"
#define ADCSPA "ADCSPA"

WiFiManager wifiManager;
String sendedHour = "xx";
String sendedDay = "xx";
int sendedMinutes = 0;

Logger logger;
RestCall restCall;
ATFunctions atFunctions;
HexFunctions hexFunctions;
Timestamp timestamp;
```

```
void sendPollCommand(String pollCommand, Logger *logger, int
timeToAttempt, int attempts = 5) {
  int currentAttempt = 1;
  String frame = "";
  String restCallResponse;
  Serial.println("Polling to SPA...");
  logger -> info(0, "Sending " + pollCommand + " to SPA.");
  String pollResponse = atFunctions.sendP2PPacket(Serial2,
pollCommand);
  String listeningResponse = atFunctions.sendATCommand(Serial2,
AT_SEMICONTINUOUS_PRECV_CONFIG_SET);
  boolean frameReceived = false;
  long actualMilis = millis();
  Serial.print("Waiting response:");
  while (!frameReceived && currentAttempt < attempts){
    Serial.print(".");
    delay(200);
    if (Serial2.available() > 0) {
      String rxData = atFunctions.readSerial(Serial2);
      rxData =
hexFunctions.hexToASCII(rxData.substring(rxData.lastIndexOf(':')+1));
      Serial.print("\nReceived data: ");
      Serial.println(rxData);
      frameReceived = true;
      frame = rxData;
      String frameLog = frame;
      frameLog.replace("\"", "'");
      logger -> info(0, "Received frame data from SPA: '" + frameLog +
"'");
    }
    if (actualMilis + timeToAttempt <= millis()) {
      Serial.printf("\nResending %s command...", &pollCommand);
      logger -> error(0, "Not frame received, resending command " +
pollCommand + " to SPA." + " Attempt: " + String(currentAttempt));
      currentAttempt++;
      if (currentAttempt >= attempts) {
        Serial.println("\nMax attempts reached.");
        logger -> error(0, "Max attempts reached. Not frame received
from SPA.");
      }
      atFunctions.sendATCommand(Serial2, AT_P2P_CONFIG_TX_SET);
      atFunctions.sendP2PPacket(Serial2, pollCommand);
      atFunctions.sendATCommand(Serial2,
AT_SEMICONTINUOUS_PRECV_CONFIG_SET);
      actualMilis = millis();
    }
  }
  if(!frame.startsWith(ERROR)) {
    if(!pollCommand.startsWith(IRR_COMMAND)) {
      restCallResponse = restCall.sendFrameData(frame, STATION_TABLE,
3);
      logger -> log(restCall.getResponseCode(), restCallResponse,
restCall.getDebugLevel(), RXSPA);
    } else {
```

```
      restCallResponse = restCall.sendFrameData(frame.substring(0,
frame.indexOf(ETC)-2) + FRAME_CLOSE, STATION_TABLE, 3);
      logger -> log(restCall.getResponseCode(), restCallResponse,
restCall.getDebugLevel(), RXSPA);

      restCallResponse = restCall.sendFrameData(IRR_PREFIX +
frame.substring(frame.indexOf(ETC), frame.indexOf(WET_WEIGHT)-2) +
FRAME_CLOSE, ETC_TABLE, 3);
      logger -> log(restCall.getResponseCode(), restCallResponse,
restCall.getDebugLevel(), RXSPA);

      restCallResponse = restCall.sendFrameData(IRR_PREFIX +
frame.substring(frame.indexOf(WET_WEIGHT), frame.length()),
WET_WEIGHT_TABLE, 3);
      logger -> log(restCall.getResponseCode(), restCallResponse,
restCall.getDebugLevel(), RXSPA);
    }
  } else {
    logger -> error(0, frame, ADCSPA);
  }
}

void pollAlarm() {
  sendPollCommand(POLL_COMMAND, &logger, 10000);
}

void irrAlarm() {
  sendPollCommand(restCall.getWeight(IRR_COMMAND), &logger, 50000);
}


void setup() {
  // Internal clock

  Serial.begin(115200);
  Serial2.begin(115200);
  wifiManager.autoConnect();
  logger.config(String(API_URL) + String(LOG_CONTEXT),
String(DB_USER), String(DB_PASS));
  restCall.config(String(API_URL), String(DB_USER), String(DB_PASS));

  sntp_setoperatingmode(SNTP_OPMODE_POLL);
  sntp_setservername(0, POOL_NTP_URL);
  sntp_init();
  sntp_sync_time(0);

  String atCommandResetResponse = atFunctions.sendATCommand(Serial2,
AT_RESET);
  String atConfigSetP2PResponse = atFunctions.sendATCommand(Serial2,
AT_P2P_CONFIG_SET);
  String atConfigGetP2PResponse = atFunctions.sendATCommand(Serial2,
AT_P2P_CONFIG_GET);
  String atConfigContRecvResponse = atFunctions.sendATCommand(Serial2,
AT_P2P_CONFIG_TX_SET);
}
```

```
void loop() {
  String day = timestamp.getDay();
  String hour = timestamp.getHours();
  String minutes = timestamp.getMinutes();
  String seconds = timestamp.getSeconds();

  if (minutes.equals("55")) {
    Serial.print("Ping response: ");
    Serial.println(restCall.ping(5));
    Serial.print("Current timestamp: ");
    Serial.println(timestamp.getLocalTimeStamp());
    delay(60000);
  }

  // Execute irrAlarm once a day at 00 hs
  if (hour.equals(IRR_HOUR) && !sendedDay.equals(day)) {
    sendedDay = day;
    sendedHour = hour;
    irrAlarm();
  }

  // Execute pollAlarm every one hour
  if (minutes.equals(POLL_MINUTES) && !hour.equals(sendedHour)) {
    sendedHour = hour;
    pollAlarm();
  }
}
```