INTERNSHIP PROJECT REPORT- 02

Name: Keerathan D

Program: IOT Mentorship

Topic: IOT based Health monitoring system using ESP 32

Mentioned Tasks of this project:

Task 1: System Setup
>> Build an IoT-based health monitoring system using ESP32, sensors, and cloud integration.

Task 2: Sensor Integration
>> Interface and the sensors - Measure Temperature
>> Ensure accuracy in real-time data capture.

Task 3: Cloud Connectivity
>> Connect ESP32 to Wi-Fi.
>> Upload data to a cloud platform (Blynk)
>> Display real-time health data on a mobile app (blynk mobile app) or web dashboard.

Task 4: Code Documentation and Testing
>> Include well-commented code.
>> Structure the code using functions for better readability.
>> Test the system in various scenarios and record observations

INDEX

ABSTRACT OF THE IOT BASED HEALTH MONITORING SYSTEM USING ESP 32

The increasing prevalence of chronic diseases and the growing elderly population necessitate a shift from traditional, episodic healthcare to continuous, remote patient monitoring. This project details the design and implementation of an IoT-based Health Monitoring System to address this need. The **overview** of the system involves a network of non-invasive wearable sensors (such as ECG, SpO2, and temperature sensors) that collect a patient's vital signs in real-time. This data is aggregated by a microcontroller unit (e.g., ESP32) and securely transmitted to a cloud platform via Wi-Fi.

The primary **objectives** of this project are:

(1) To design and develop a low-cost, portable, and user-friendly prototype for real-time monitoring of vital health parameters.

(2) To implement a secure cloud-based backend for data storage, processing, and analysis.

(3) To create an alert mechanism that automatically notifies healthcare providers (doctors, nurses) or family members via an application or SMS during emergency situations, such as a sudden fall or critical vital readings.

(4) To provide an accessible web or mobile-based dashboard for remote visualization of patient data by authorized medical personnel.

The expected **outcomes** of this system are a significant improvement in the quality of remote patient care, enabling timely medical intervention and reducing the frequency of hospital readmissions. This system will enhance patient safety and independence, particularly for elderly individuals living alone. Furthermore, it will reduce the burden on healthcare infrastructure by facilitating efficient remote consultations and create a valuable dataset for future medical research and predictive health analytics.

Background and Motivation

Modern healthcare is at a critical inflection point. We face a "perfect storm" of challenges: a rapidly aging global population, a significant increase in the prevalence of chronic diseases such as diabetes, hypertension, and cardiovascular conditions, and the spiraling costs of in-hospital care. The traditional healthcare model, which relies on episodic, in-person visits to a clinic or hospital, is reactive by nature. It is often insufficient for managing chronic conditions that require continuous oversight.

The limitations of this traditional model are clear. Monitoring is inconvenient for patients, requiring travel and appointments. It is expensive for the healthcare system, consuming valuable hospital resources and staff time. Most importantly, it is episodic; a patient's vital signs are only captured for a few minutes during a visit, missing critical fluctuations that may occur throughout the day. This can lead to delayed diagnoses, missed warning signs, and preventable emergencies.

This is where the Internet of Things (IoT) emerges as a transformative solution. IoT, specifically the Internet of Medical Things (IoMT), refers to the network of interconnected medical devices, sensors, and healthcare IT systems. By equipping patients with small, wearable sensors, we can capture a continuous stream of physiological data. This data can be transmitted to the cloud in real-time, analyzed, and made accessible to medical professionals from anywhere in the world. This paradigm shift from reactive, episodic care to proactive, continuous monitoring promises to revolutionize patient outcomes and healthcare efficiency.

There is a critical need for a continuous, reliable, and low-cost system that allows for the remote monitoring of patients' vital signs. Such a system must be able to enable the early detection of health deterioration through real-time data analysis and provide an immediate, automated alert mechanism to trigger timely medical intervention. This is particularly crucial for high-risk individuals, such as the elderly living alone or patients recently discharged from the hospital, for whom a sudden health event could be catastrophic if not addressed immediately.

This project proposes the design, development, and implementation of an IoT-based Real-Time Health Monitoring System. The core idea of this solution is to create a wearable, sensor-based system that captures key vital signs—specifically body temperature, heart rate, and blood oxygen saturation (SpO2).

This data is collected by a central microcontroller, which then securely transmits the readings over a Wi-Fi network to a cloud-based platform. On the cloud, the data is stored, processed, and analyzed against predefined thresholds. If any parameter falls into a critical range, an automated alert is sent to a registered healthcare provider or family member. Finally, all real-time and historical data is presented on an intuitive, secure web-based dashboard, allowing doctors to remotely monitor their patients' health status, identify trends, and make informed clinical decisions.

Objectives

Primary Objectives The fundamental goals that must be achieved for this project to be considered a success are:

1. **To design and develop a functional hardware prototype** of an IoT-based health monitoring system that is portable, non-invasive, and can be worn by a user.
2. **To accurately collect and transmit multiple vital signs** in real-time. This specifically includes body temperature (in °C/°F), heart rate (in BPM), and blood oxygen saturation (SpO2 as a percentage).
3. **To implement a scalable and secure cloud-based backend** for reliable data ingestion, time-series storage, and processing of the incoming sensor data.

Secondary Objectives These objectives build upon the primary ones to add critical features and value to the system:

1. **To develop a threshold-based alert system** that automatically sends a notification (e.g., email, SMS, or app notification) when any of the measured vital signs cross a predefined "unsafe" limit (e.g., heart rate > 120 BPM or SpO2 < 90%).

2. **To create a user-friendly web or mobile application** that provides a secure login for authorized personnel (doctors, caregivers) to remotely visualize the patient's real-time data and historical trends through graphs and charts.

3. **To ensure the system is low-cost**, utilizing readily available and affordable electronic components, and energy-efficient to allow for extended monitoring periods on a single battery charge.

4. **To validate the system's accuracy** by comparing its readings against standard, off-the-shelf medical instruments (e.g., a commercial pulse oximeter and thermometer) to ensure data reliability.

Outcome

In-Scope: The primary focus of this project is to develop a functional prototype that successfully demonstrates the end-to-end data flow. This includes:

- Monitoring three specific vital signs: body temperature, heart rate, and SpO2.
- Focusing on a single user/patient per device.
- Transmitting data over Wi-Fi to a cloud platform.
- Storing the data in a time-series database.
- Implementing a basic threshold-based alert system.
- Displaying the data on a web-based dashboard.

Out-of-Scope: This project has several important limitations.

- The developed prototype is **not a certified medical device** and is not intended for formal medical diagnosis, treatment, or use in life-support situations. It is a proof-of-concept.
- The project will not undergo the rigorous clinical trials or regulatory approval (e.g., FDA, HIPAA compliance) required for a commercial medical product, though security best practices will be considered.

- Advanced features such as predictive analytics using machine learning are considered future work.
- Monitoring will be limited to the specified sensors; other vitals like blood pressure or glucose are not included.

Functional outcomes

- Continuous vital-sign capture using commodity sensors on ESP32, with on-device preprocessing and real-time visualization on an embedded web interface or LCD/OLED, confirming on-node situational awareness even without the internet.
- Remote access to the same streams via MQTT/cloud or app frameworks, enabling caregivers to monitor trends and receive alerts when readings cross configured thresholds (e.g., fever or hypoxemia), which literature links to timelier interventions and improved monitoring coverage.
- Data retention and export from cloud dashboards (e.g., ThingSpeak/DB backends) for longitudinal analysis, supporting basic analytics and future model building for anomaly or event prediction in telehealth workflows

INTRODUCTION TO THE IOT BASED HEALTH MONITORING SYSTEM

A robust introduction to an ESP32-based IoT health monitoring system should establish the clinical motivation, survey foundational work, and frame the system's architecture and capabilities from sensor to cloud, culminating in clear problem–solution alignment for remote, continuous care settings. The approach centers on affordable sensors connected to an ESP32 microcontroller that acquires temperature, heart rate, and SpO_2, renders a local dashboard, and optionally streams data to cloud or broker services for alerts and longitudinal insight in home and community care scenarios

Healthcare delivery increasingly depends on timely recognition of physiological change, yet conventional, intermittent spot checks leave gaps that can delay intervention; IoT monitoring addresses this by producing continuous streams that support earlier, data-driven responses and reduce caregiver burden. Recent ESP32 projects show practical paths to implement this vision at low cost, hosting local web pages and publishing to remote dashboards for real-time review of vitals such as body temperature, heart rate, and blood oxygen saturation. The low barrier to entry of ESP32 hardware and open software makes such systems feasible for students, researchers, and resource-constrained clinics, expanding access to remote observation beyond proprietary devices.

ESP32 integrates dual-core processing with Wi-Fi/Bluetooth, enabling deterministic sensor sampling while serving a web UI or maintaining MQTT connections without additional gateways in many scenarios. Common sensor choices include DS18B20 for body temperature and MAX3010x series for pulse oximetry and heart rate, which connect via OneWire and I^2C respectively and are well supported by mature open-source libraries for embedded development. Tutorials and reference builds demonstrate end-to-end wiring and code patterns—reading sensors, formatting JSON, and exposing HTTP endpoints—so that even first prototypes can deliver stable, near-real-time telemetry to browsers on the same network

Systematic and scoping reviews of IoT in healthcare consistently describe three-layer architectures—perception (sensors), network (Wi-Fi/MQTT/HTTP), and application (dashboards/alerts)—that improve monitoring continuity and facilitate remote clinical oversight. Academic and practitioner projects with ESP32 validate these layers: collecting vitals, publishing to mobile apps or cloud backends, and raising threshold-based notifications that support home, elderly, and chronic-care use cases. Hands-on build guides further exemplify how a single ESP32 node can host a responsive web interface, allowing users to view live vitals locally while optionally forwarding the same data to remote services for history and sharing with clinicians.

- Patients outside clinical settings often lack continuous vital-sign monitoring, leading to delayed recognition of fever, hypoxemia, or tachycardia; a low-cost, connected solution is needed to close this gap.
- Existing medical-grade devices can be expensive or closed, limiting customization and integration for research and education; an ESP32-based design enables openness and rapid iteration while achieving practical accuracy for remote observation and alerting in pilot contexts.
- Care teams require simple dashboards and dependable alerts rather than raw sensor streams; the system must translate vitals into actionable notifications with minimal latency and intuitive visualization on web or mobile clients.

The project proposes a modular ESP32 node that reads temperature and PPG-based HR/SpO_2 at fixed intervals, filters noise, and exposes data through two complementary paths: a local web server for immediate visualization and an MQTT/HTTP channel for remote dashboards and notifications. The architecture follows established IoT healthcare patterns—sensor acquisition, edge processing, network transport, storage/analytics, and user interface—ensuring portability from a single bedside device to multi-node deployments with shared infrastructure. A rules engine evaluates thresholds (for example, temp > 38 °C or $SpO_2 < 90\%$) and raises on-page and push alerts, aligning with common remote-monitoring protocols in prior ESP32 implementations and studies using mobile platforms like Blynk or custom web apps

Expected capabilities

A single device can serve a browser-based dashboard showing live vitals and status, useful for bedside or household monitoring without internet access, while the same firmware can publish to cloud or LAN brokers for remote oversight and historical analytics when connectivity is available. Prior ESP32 research demonstrates that this dual-path model—local UI plus remote streaming—offers both resilience and reach, allowing users to continue viewing readings during outages and backfilling the cloud once the connection is restored. In aggregate, such systems contribute to improved vigilance in home care, offering evidence-based support for early intervention and workload reduction reported across IoT monitoring literature.

Significance and scope

The introduction frames the project as an accessible, evidence-aligned response to real gaps in outpatient monitoring, emphasizing affordability, openness, and adaptability across single-patient and multi-patient scenarios. By grounding the design in ESP32 best practices and widely used sensor modules, the project ensures reproducibility for coursework and research while laying the groundwork for extensions like encryption, mobile notifications, and predictive analytics as future work. Practical build guides attest to feasibility, making this a strong candidate for lab validation and pilot deployments in educational or community health settings where rapid, low-cost innovation is essential.

METHODOLOGY

A practical methodology for the ESP32 IoT health monitoring system combines hardware integration, edge firmware, networking, data/UI layers, and validation, with Blynk used as the cloud/app layer for visualization, notifications, and remote control via virtual pins and datastreams mapped to dashboard widgets. The Blynk dashboard shown can be described in terms of datastreams (e.g., V-pins), widget roles, ranges/units, and automations, aligning with documented Blynk patterns for gauges, value displays, and charts on ESP32 projects that send vitals like temperature, HR, SpO_2, and BMI to the platform in real time

- System design: adopt a layered architecture—sensing (DS18B20, MAX3010x), edge compute/communications (ESP32), transport (Wi-Fi HTTP/MQTT to Blynk), application (Blynk Cloud dashboards, notifications), and storage/analytics—consistent with IoT healthcare frameworks that improve monitoring continuity and response.
- Platform selection: use Blynk Cloud for device provisioning, datastreams, dashboards, and automations; ESP32 is supported with reference templates and code patterns for virtual writes/reads to sync telemetry and controls.

Firmware design (ESP32)

Initialization: include TemplateID, DeviceName, and Auth Token; connect to Wi-Fi; start Blynk client; configure timers for periodic sensor sampling and cloud updates using BlynkTimer to avoid blocking loops.

Datastream mapping: publish sensor values with Blynk.virtualWrite(Vx, value) to assigned virtual pins whose types/ranges match widget datastreams; this mirrors common examples that send temperature, humidity, BPM, SpO_2, and body temperature to V3–V7, etc..

Input handling: implement BLYNK_WRITE(Vx) callbacks to receive user-entered parameters (e.g., height/weight) and compute derived metrics like BMI on the device or server; Blynk documents handler patterns for parsing types safely

- Communication model: ESP32 pushes values to Blynk Cloud at controlled intervals; avoid flooding by sending only on change or at timer ticks to preserve connection health, per Blynk’s guidance on virtualWrite usage and message limits.

- Provisioning/templates: use Blynk Templates to predefine datastreams (name, V-pin, type, min/max, units) and then attach widgets; Blynk Blueprints illustrate this workflow, including threshold notifications via automations

Blynk dashboard analysis

- Temperature gauge: a radial Gauge shows 31 °C with a range of 0–121; this is a standard Blynk Gauge bound to a virtual datastream (e.g., V0) whose min/max and units are defined in the Template's Datastream tab; the gauge fills proportionally and reflects the latest value received from the device.
- BMI card: a Value Display widget shows BMI 24.22; typically this is populated either by on-device computation Blynk.virtualWrite(Vx, bmi) or by server-side rules; BMI uses weight and height inputs bound to separate datastreams and displayed as a numeric card with configured decimals and unitless label per Blynk's Value Display conventions.
- User height in cm: a control widget with increment/decrement buttons is likely bound to a virtual datastream (e.g., Vh) that the device reads via BLYNK_WRITE, storing the user's target height; the "current height" card 170 cm appears as a Value/Labeled Value widget bound to a separate read-only datastream updated by the device after validation, which is a common pattern to keep "input" and "current" distinct.
- User weight in kg and current weight: the same pattern applies to weight—an input widget to adjust or submit weight, and a read-only display reflecting the latest device-accepted value; this enables BMI recomputation on the device and immediate UI feedback through bound datastreams.
- Time range controls: the "Live/1h/6h/1d/…" selector indicates that certain widgets (Charts) can show historical trends when datastream history is enabled; even when a card shows a single value, Blynk supports historical visualization via Chart widgets tied to the same V-pin, as demonstrated in ESP32+DS18B20 tutorials.
- Edit and indicators: the "Online" status confirms active device-cloud connection; the "Edit" mode allows changing widget bindings and datastream parameters; this is standard Blynk web dashboard behavior for template-based projects

Proposed System Architecture

The system is designed as a four-layer architecture, which is a standard model for IoT applications. This layered approach ensures modularity, scalability, and a clear separation of concerns.

1. **Sensor Layer (Data Acquisition):** This is the physical layer that interfaces directly with the patient. Its sole responsibility is to capture raw physiological data.
   - **Sensors:** We will use a DS18B20 waterproof temperature sensor for its accuracy in measuring body temperature. For heart rate and blood oxygen, a MAX30100 or similar pulse oximeter sensor will be used. This sensor works by shining light through a fingertip and measuring the absorption.
   - *(Optional Additions):* For a more advanced system, an AD8232 ECG sensor could be added to capture heart rhythm, and an MPU6050 accelerometer/gyroscope could be integrated for fall detection.
2. **Gateway Layer (Data Processing & Transmission):** This layer acts as the "brain" of the wearable device.
   - **Microcontroller:** An **ESP32** microcontroller is the ideal choice for this project. It is low-cost, powerful, and has built-in Wi-Fi and Bluetooth capabilities, eliminating the need for a separate communication module. It has sufficient analog and digital pins to interface with all the sensors.
   - **Function:** The ESP32 will run embedded code (written in C++ via the Arduino IDE) to periodically poll the sensors, read the data, perform minor processing (e.g., averaging readings), format the data into a standardized format (like JSON), and then transmit it securely over Wi-Fi to the cloud layer.
3. **Cloud Layer (Data Storage & Analysis):** This is the high-performance, scalable backend of the system.
   - **Platform:** We will use **Firebase (a Google platform)**. It provides a real-time NoSQL database (Firestore) perfect for storing sensor data, secure authentication for users (doctors/patients), and cloud functions for running backend logic.

- **Function:** The Firestore database will store all incoming sensor readings, each with a patient ID and a timestamp. A **Cloud Function** will be set up to trigger every time new data is written to the database. This function will analyze the data, check it against the patient's predefined safety thresholds, and if a threshold is breached, it will use a service like Twilio (for SMS) or SendGrid (for email) to dispatch an alert.

4. **Application Layer (Data Visualization):** This is the user-facing layer and the system's main interface.
    - **Interface:** A web-based dashboard will be developed using modern web technologies (HTML, CSS, and JavaScript with a framework like React).
    - **Function:** A doctor or caregiver will log in using their secure credentials (managed by Firebase Authentication). After logging in, they will see a list of their patients. Upon selecting a patient, the dashboard will fetch the relevant data from Firestore and display the live vital signs, along with interactive graphs showing historical trends for the last 24 hours, week, or month.

Automations and alerts

- Temperature alert: create an automation "if temperature > 38 °C then notify" to push mobile alerts; Blynk Blueprints illustrate configuring threshold notifications tied to datastreams and units.

- BMI bands: optional automation can tag BMI ranges and change widget color/state; while BMI is not a clinical diagnostic, it can be used for demonstration of rule-based UI changes using widget properties and server-side automations.

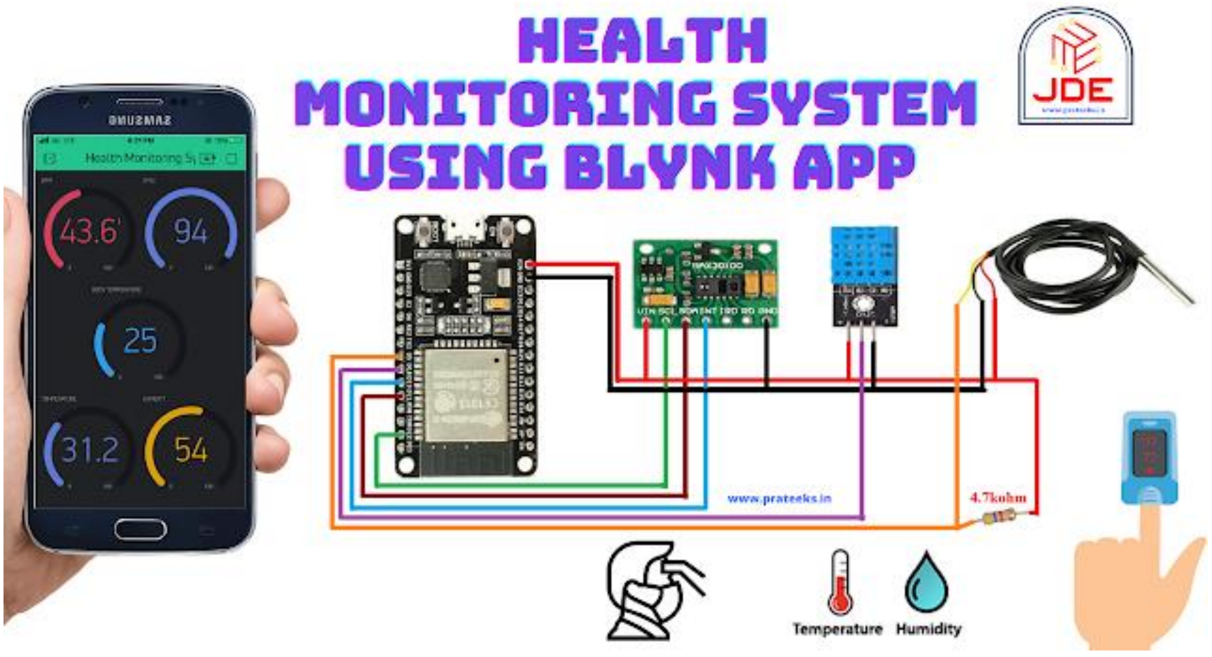
Deployment and management

- Template-based provisioning: define datastreams once, then add multiple devices for different users/patients; Blynk supports remote device management and OTA via templates and projects for ESP32 fleets.

- Rate limiting and stability: follow Blynk guidance to avoid message floods by using timers and "on change" publishing; ensure widgets and datastreams have sensible min/max/decimals to prevent UI jitter and unit confusion.

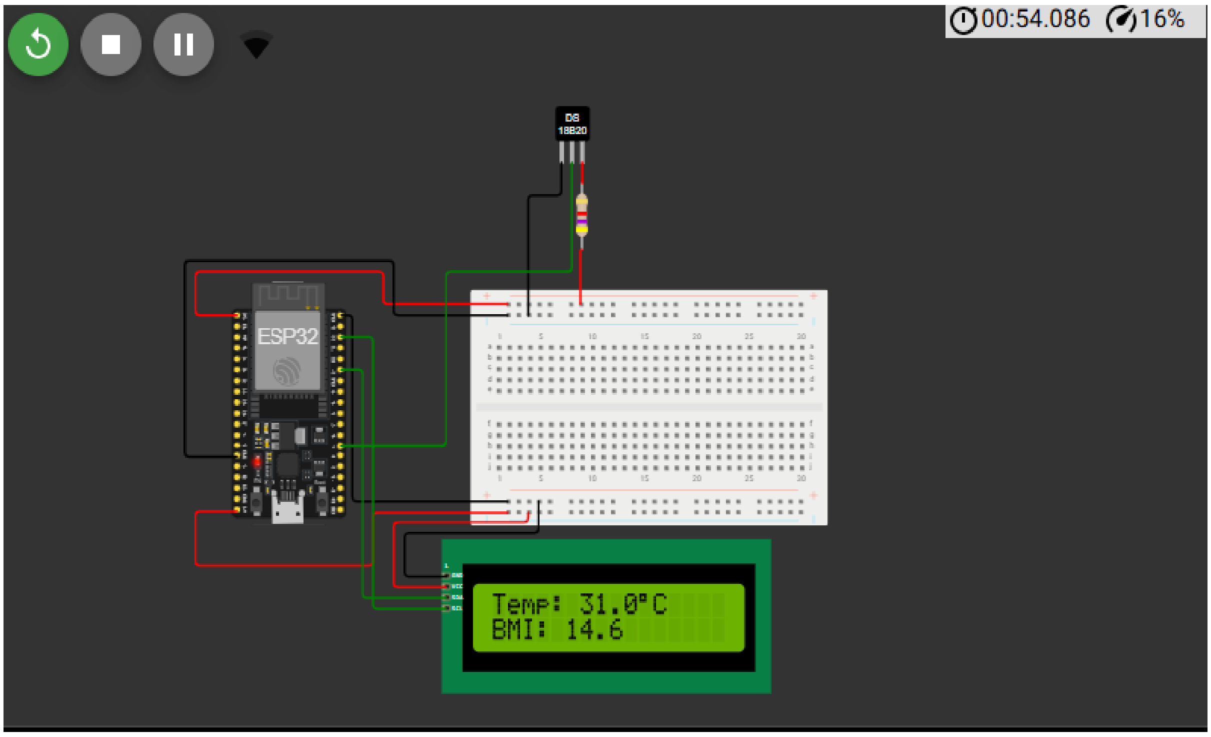
Traceability to prior art

- End-to-end ESP32+Blynk health monitoring builds have demonstrated the same architecture—sensing, virtual pin mapping, dashboards, and alerts—making the proposed methodology and the observed dashboard design consistent with best-practice tutorials and articles.
- The broader healthcare IoT literature supports the layered approach and the benefits of persistent, remote telemetry for timelier interventions, which this system operationalizes through Blynk's datastreams, widgets, and automations on top of ESP32 nodes.



IMPLEMENTATION

i) The circuit diagram which is used in online simulator – wokwi

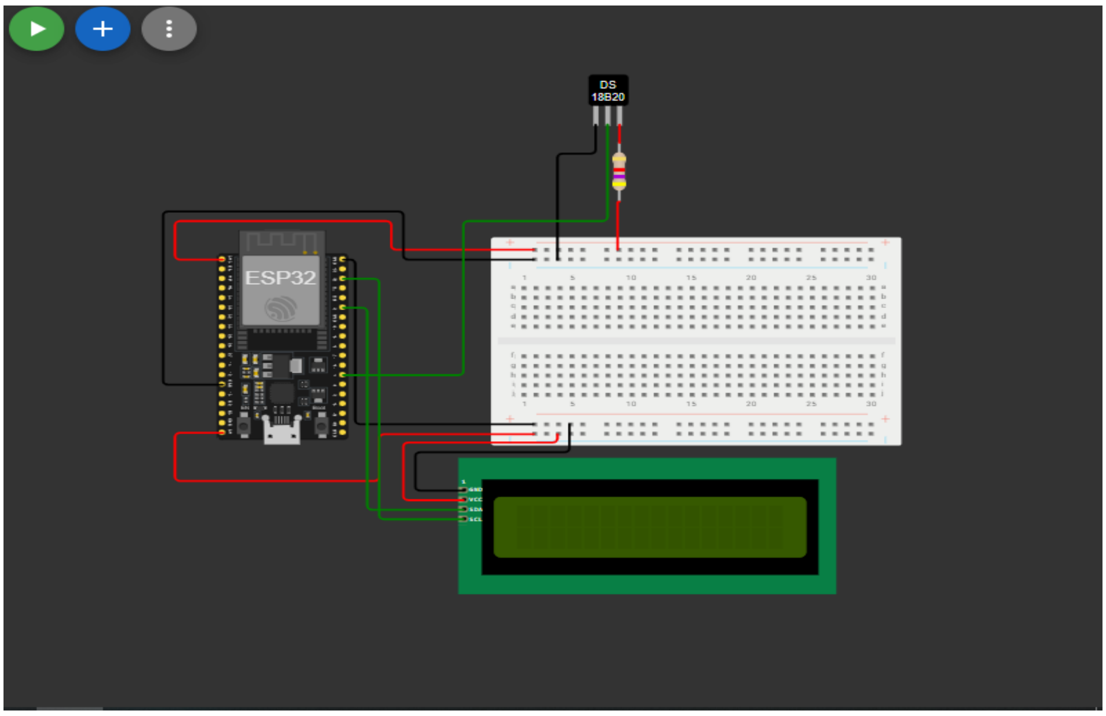


COMPONENTS USED:

- ESP32 development board (DevKit, WROOM, or C3), main MCU with Wi-Fi/BLE for sensing and cloud connectivity.
- DS18B20 digital temperature sensor (waterproof or TO-92 package) for body temperature; requires a 4.7 kΩ pull-up resistor on the data line.
- MAX30100/MAX30102/MAX30105 pulse-oximeter module for heart rate and SpO_2 via I^2C, widely used with ESP32 health projects.
- DHT11/DHT22 or BME280 ambient sensor for room temperature/humidity context and calibration support.
- 4.7 kΩ resistor for DS18B20 OneWire bus pull-up to VCC as per standard wiring guidance.

- Breadboard and male-male jumper wires for prototyping interconnections between sensors and MCU.
- USB cable and 5 V power source for ESP32 power and programming via Arduino IDE/PlatformIO.
- Optional 16x2 LCD or small OLED display for local readouts independent of the phone/cloud.
- Optional DC-DC buck module (e.g., LM2596) for stable power if using batteries or higher-voltage inputs.
- Smartphone and Blynk Cloud account/app for dashboards, datastreams, notifications, and device management.
- Arduino IDE/PlatformIO toolchain with libraries for DS18B20/OneWire and MAX3010x, plus Blynk device template configuration for virtual pins.
- Optional enclosure, cables, and patient sensor mounts for practical deployment and motion artifact reduction with the PPG sensor

ii) before simulation



Working:

1) Prepare tools and accounts

- Install Arduino IDE or PlatformIO and add the ESP32 boards package; confirm you can select your ESP32 board and a serial port for flashing.
- Create a Blynk account, create a Template for the project, and note the Template ID, Device Name, and Auth Token used by the firmware to connect to Blynk Cloud.

2) Create Blynk datastreams

- In the Template's Datastreams tab, add Virtual Pin streams with proper types/units, for example: V0 temperature (Double, °C), V1 height input (Double, cm), V2 weight input (Double, kg), V3 BMI (Double, no unit); this follows Blynk's documented Virtual Pin setup flow.
- If needed, add Enum or on/off datastreams for controls/automations, and set sensible min/max and decimals so widgets show stable values per Blynk guidance.

3) Assemble the hardware

- Wire the DS18B20: VCC → 3.3 V, GND → GND, DATA → any free GPIO (e.g., GPIO 5), and place a 4.7 kΩ pull-up resistor from DATA to VCC, which is the standard OneWire requirement on ESP32.
- Optionally connect MAX3010x pulse-oximeter via I^2C for HR/SpO_2 and an LCD/OLED for local display; these are commonly paired with ESP32 in health projects and can be added later without changing the cloud flow.

4) Build the ESP32 firmware

- Include libraries: WiFi, BlynkSimpleEsp32, OneWire, DallasTemperature; initialize Serial, sensors, and Wi-Fi, then start Blynk with Template ID/Device Name/Auth Token as in Blynk examples.
- Implement timed tasks with BlynkTimer: read DS18B20 every 1–2 seconds and publish via Blynk.virtualWrite(V0, tempC); use non-blocking code to keep the connection stable as recommended by Blynk docs and examples.

5) Handle user inputs and compute BMI

- Add BLYNK_WRITE(V1) and BLYNK_WRITE(V2) callbacks to receive height and weight from the app; store in globals and validate ranges before use, per the standard virtual pin handler pattern.
- Compute BMI = weight_kg / (height_m^2) whenever inputs change; publish to V3 with Blynk.virtualWrite(V3, bmi) so your BMI card updates in real time, mirroring common ESP32+Blynk health tutorials.

6) Design the Blynk dashboard

- Add a Gauge for temperature bound to V0 with unit °C and a realistic range; Blynk's Gauge widget renders the radial indicator and label from the datastream values.
- Add Value Display cards for BMI (V3), current height, and current weight, and Step/Buttons/Sliders bound to V1 and V2 to let users adjust inputs; Value Display widgets show labels/decimals per the configuration.
- Enable history or add a Chart for temperature to view trends across Live/1h/1d as supported in Blynk dashboards and Blueprints.

7) Flash, connect, and verify

- Enter Wi-Fi SSID/password and Blynk credentials in the sketch; flash the ESP32; confirm in Serial Monitor that Wi-Fi and Blynk sessions connect and that sensor values are read without NaNs or timeouts.
- Open the Blynk web or mobile dashboard: you should see the device status Online and widgets updating in near real time; adjust height/weight controls and confirm BMI recomputes instantly through your V-pin mapping.

8) Add thresholds and notifications

- In Blynk Automations, create rules like "if V0 temperature > 38 then Notify" to push alerts to your phone; Blueprints show the same pattern for threshold notifications tied to datastreams and units.
- For stability, publish on a timer or only when values change to avoid flooding; Blynk documentation emphasizes rate limits and virtualWrite pacing for reliable sessions.

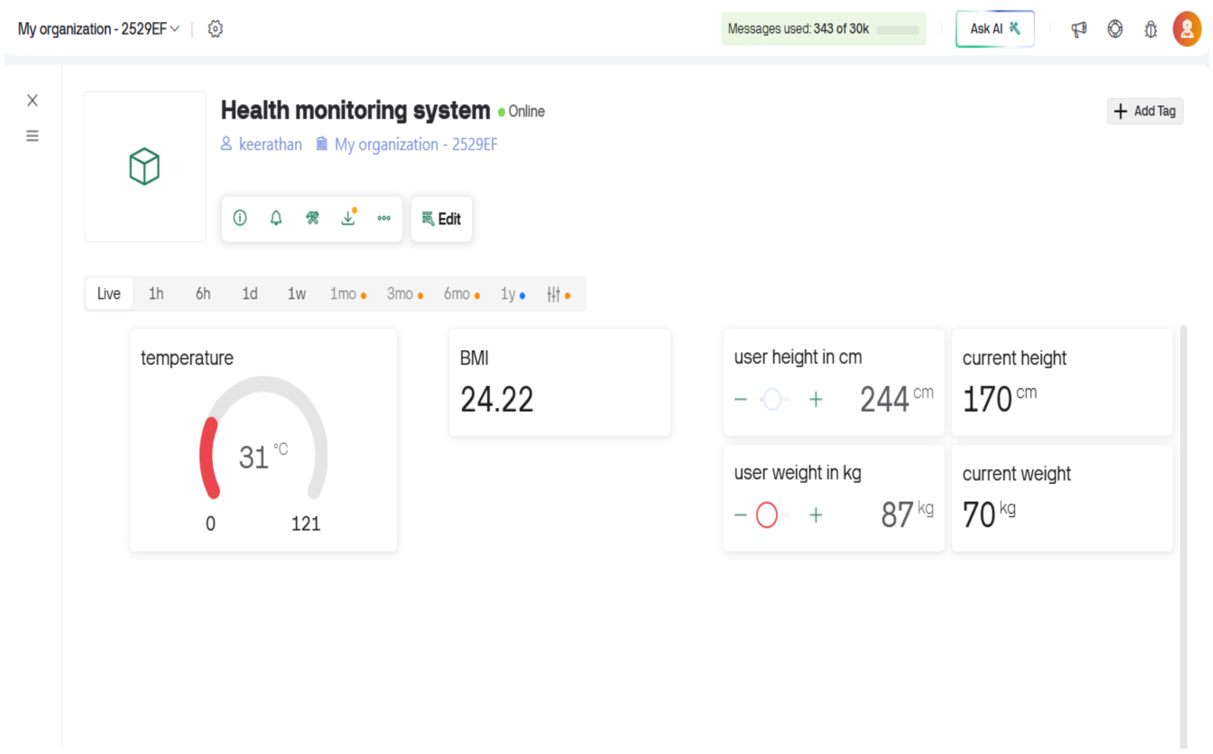
9) Extend to more vitals (optional)

- Integrate MAX3010x for HR/SpO₂ over I²C; add new datastreams (e.g., V4 HR, V5 SpO₂) and corresponding widgets; several ESP32+Blynk articles demonstrate combining DS18B20 with MAX3010x and DHT/BME sensors in one app.
- If a local web UI is desired, you can also serve an ESP32 web page in parallel for offline viewing; the web-server pattern with DS18B20 is well documented and complements cloud dashboards.

10) Test, calibrate, and harden

- Compare readings to reference thermometers/oximeters and adjust filtering or sampling intervals; DS18B20 setup and read timing patterns for stable conversions are covered in step-by-step guides.
- Add reconnect/backoff logic, verify Automations fire at thresholds, and ensure widget ranges/units are correct; this is standard practice in Blynk setups to maintain reliability and clear UI semantics

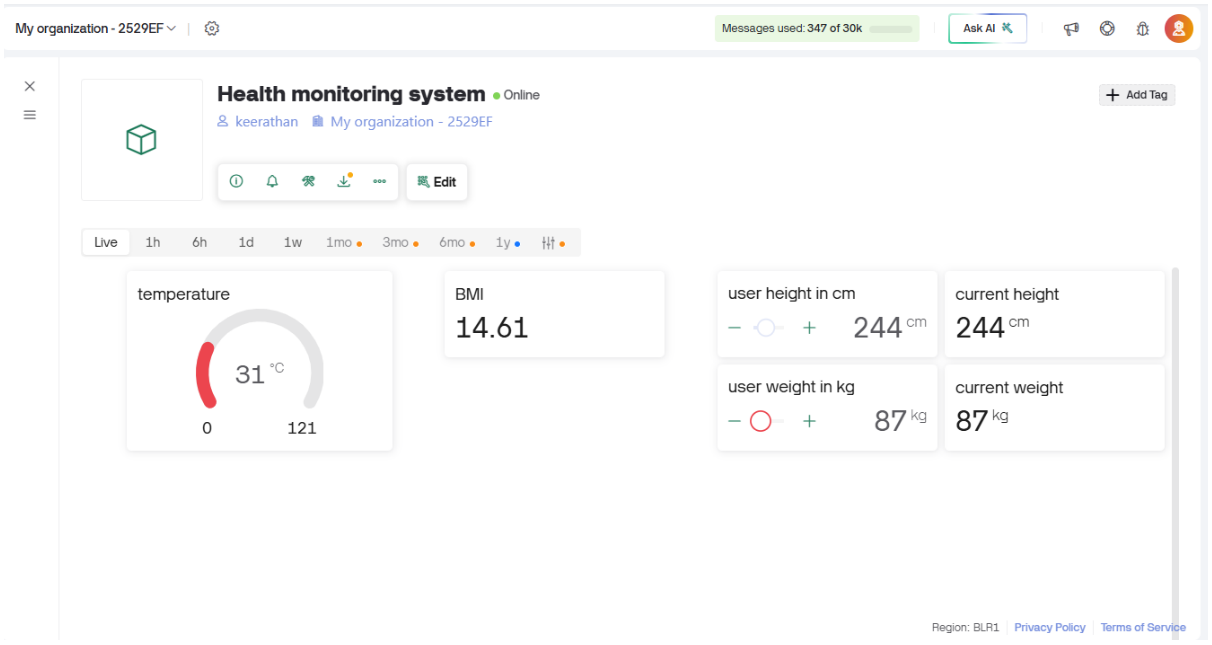
iv) dashboard view of blynk cloud platform- before



Health monitoring system Online
keerathan My organization - 2529EF

temperature 31 °C (0 – 121)
BMI 24.22
user height in cm 244 cm
current height 170 cm
user weight in kg 87 kg
current weight 70 kg

V) After the data is collected to esp 32 t from the user input widget the data is processed and shows the BMI value and changes the current state values , it even displays the temperature.



temperature 31 °C (0 – 121)

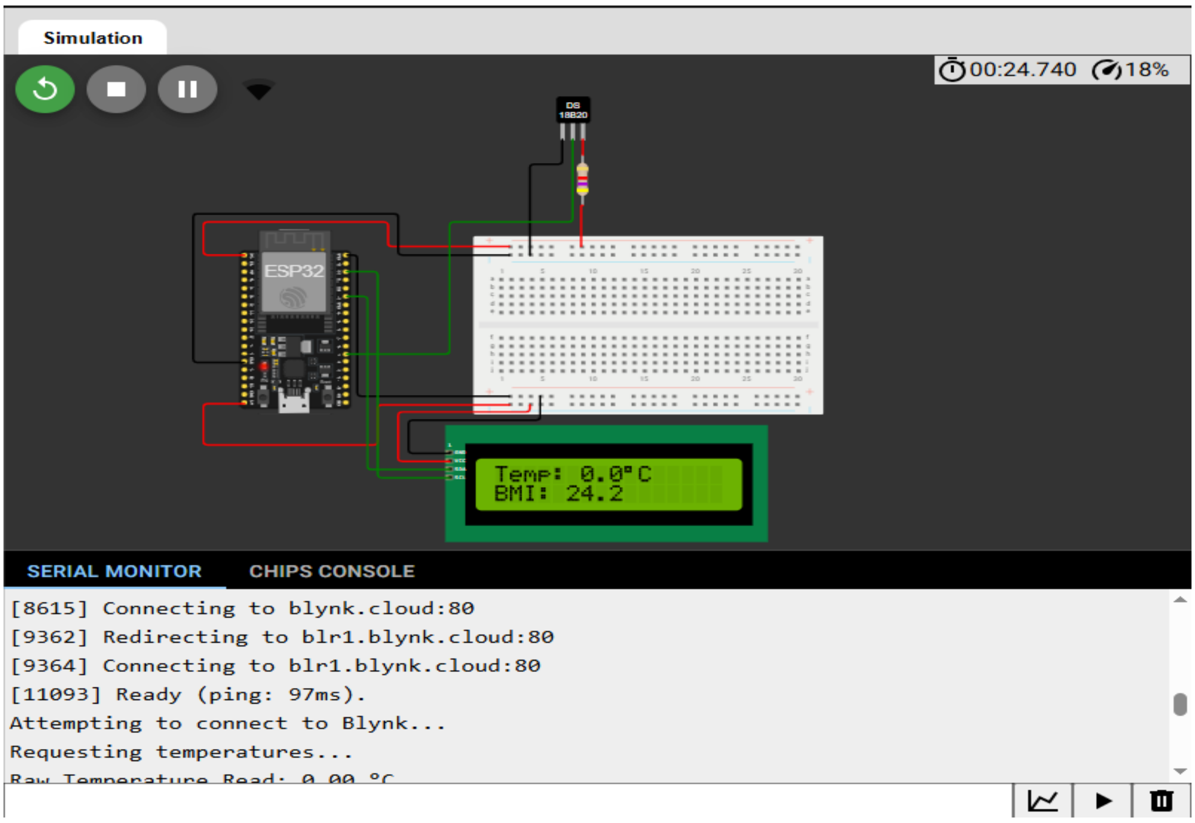
BMI 14.61

user height in cm 244 cm

current height 244 cm

user weight in kg 87 kg

current weight 87 kg



```
[8615] Connecting to blynk.cloud:80
[9362] Redirecting to blr1.blynk.cloud:80
[9364] Connecting to blr1.blynk.cloud:80
[11093] Ready (ping: 97ms).
Attempting to connect to Blynk...
Requesting temperatures...
Raw Temperature Read: 0.00 °C
```

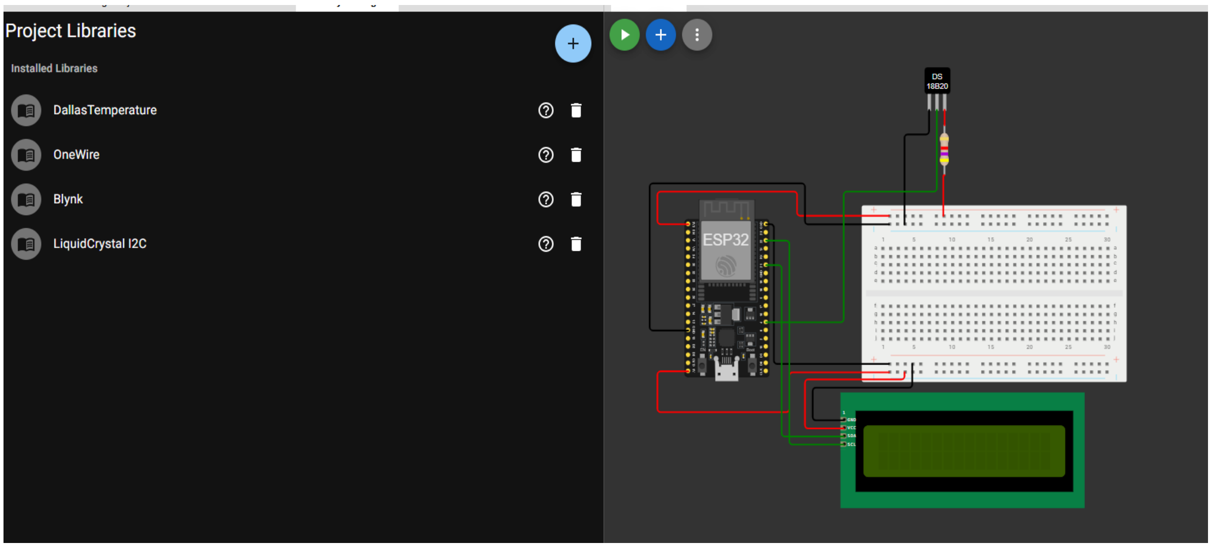
Code used for simulation:

```
#define BLYNK_PRINT Serial // IMPORTANT: Define this at the very top to enable
Serial Monitor for Blynk

// Include Wokwi's secrets file for secure storage of credentials
#include "secrets.h"

// Include necessary libraries for WiFi, Blynk, 1-Wire sensor, and I2C LCD
#include <WiFi.h>
#include <WiFiClient.h>
#include <BlynkSimpleEsp32.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <LiquidCrystal_I2C.h> // For I2C LCD communication

// --- Blynk Authentication & WiFi Credentials ---
char auth[] = BLYNK_AUTH_TOKEN;
char ssid[] = WIFI_SSID;
char pass[] = WIFI_PASS;

// --- DS18B20 Temperature Sensor Setup ---
#define ONE_WIRE_BUS 4 // Connected to ESP32 GPIO 4 (matching your Wokwi
diagram)
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

// --- I2C LCD Display Setup ---
LiquidCrystal_I2C lcd(0x27, 16, 2); // Address 0x27, 16 columns, 2 rows

// --- Global Variables for Health Data ---
float currentTemperature = 0.0;
float userHeight_cm = 170.0;
float userWeight_kg = 70.0;
float bmi = 0.0;

// --- Blynk Timer Object ---
BlynkTimer timer;

// --- Function to update the local LCD display ---
void updateLcdDisplay() {
  lcd.clear(); // Clear the entire LCD screen before writing new data

  // Display Temperature on the first line of the LCD
  lcd.setCursor(0, 0); // Set cursor to column 0, row 0
  lcd.print("Temp: ");
  if (currentTemperature == DEVICE_DISCONNECTED_C) { // Check for sensor error
    lcd.print("Error!");
  } else {
    lcd.print(currentTemperature, 1); // Print temperature with 1 decimal
place
    lcd.print((char)223);             // Print the degree symbol (ASCII code
223)
    lcd.print("C");                   // Print 'C' for Celsius
  }

  // Display BMI on the second line of the LCD
  lcd.setCursor(0, 1); // Set cursor to column 0, row 1
  lcd.print("BMI: ");
  lcd.print(bmi, 1); // Print BMI with 1 decimal place
}

// --- Function to read temperature from the DS18B20 sensor ---
void getTemperature() {
  Serial.println("Requesting temperatures...");
  sensors.requestTemperatures(); // Send the command to get temperatures
  currentTemperature = sensors.getTempCByIndex(0); // Read temperature in
Celsius from the first sensor found

  // --- DEBUGGING OUTPUT FOR TEMPERATURE ---
  if (currentTemperature == DEVICE_DISCONNECTED_C) {
    Serial.println("ERROR: DS18B20 sensor not detected or disconnected!");
    // If sensor is disconnected, we might want to send 0 or a specific error
code to Blynk,
    // or simply not send anything. For now, we'll keep the value as is.
  } else if (currentTemperature == -127.00) { // Specific error value for
DallasTemperature
    Serial.println("ERROR: DS18B20 returned invalid temperature (-127.00).
Check wiring.");
  }

  Serial.print("Raw Temperature Read: ");
  Serial.print(currentTemperature);
  Serial.println(" °C");

  // Only send valid temperature data to Blynk
  if (currentTemperature != DEVICE_DISCONNECTED_C && currentTemperature != -
127.00) {
    Blynk.virtualWrite(V1, currentTemperature); // Send to Blynk V1
    Serial.println("Temperature sent to Blynk V1.");
  } else {
    Blynk.virtualWrite(V1, 0.0); // Send 0 or a placeholder to Blynk if
there's an error
    Serial.println("Temperature not sent to Blynk due to sensor error.");
  }
  // Update the local LCD display with the latest data (will show "Error!" if
disconnected)
  updateLcdDisplay();
}

// --- Function to calculate BMI based on userHeight_cm and userWeight_kg ---
void calculateBMI() {
  if (userHeight_cm > 0 && userWeight_kg > 0) {
    float height_m = userHeight_cm / 100.0;
    bmi = userWeight_kg / (height_m * height_m);
    Serial.print("BMI: ");
    Serial.println(bmi);
  } else {
    bmi = 0.0;
    Serial.println("Invalid Height or Weight for BMI calculation. Set to 0.");
  }

  Blynk.virtualWrite(V2, bmi);
  Blynk.virtualWrite(V3, userHeight_cm);
  Blynk.virtualWrite(V4, userWeight_kg);
  Serial.println("BMI, Height, Weight sent to Blynk.");

  updateLcdDisplay();
}

// --- Blynk callback function: triggered when data is sent to Virtual Pin V5
(Height Input) ---
BLYNK_WRITE(V5) {
  userHeight_cm = param.asFloat();
  Serial.print("Received Height (V5): ");
  Serial.println(userHeight_cm);
  calculateBMI();
}

// --- Blynk callback function: triggered when data is sent to Virtual Pin V6
(Weight Input) ---
BLYNK_WRITE(V6) {
  userWeight_kg = param.asFloat();
  Serial.print("Received Weight (V6): ");
  Serial.println(userWeight_kg);
  calculateBMI();
}

// --- Setup Function: Runs once when the ESP32 starts ---
void setup() {
  Serial.begin(115200); // Initialize serial communication for debugging

  sensors.begin();      // Start the DS18B20 sensor library
  // --- DEBUGGING: Check for DS18B20 presence ---
  Serial.print("Locating DS18B20 devices...");
  Serial.print("Found ");
  Serial.print(sensors.getDeviceCount(), DEC);
  Serial.println(" devices.");
  if (sensors.getDeviceCount() == 0) {
    Serial.println("CRITICAL: No DS18B20 sensor found! Please check wiring.");
  }

  // Initialize the I2C LCD
  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Starting...");
  lcd.setCursor(0, 1);
  lcd.print("Connecting to WiFi");

  // Connect to the Blynk server
  Blynk.begin(auth, ssid, pass);
  Serial.println("Attempting to connect to Blynk...");

  // Send initial default height and weight to Blynk V3 and V4
  Blynk.virtualWrite(V3, userHeight_cm);
  Blynk.virtualWrite(V4, userWeight_kg);

  // Configure the Blynk timer to call getTemperature() and calculateBMI()
every 5000 milliseconds (5 seconds)
  timer.setInterval(5000L, getTemperature);
  timer.setInterval(5000L, calculateBMI);

  // Schedule an immediate (after a small delay) initial BMI calculation and
temp read
  timer.setTimeout(1000L, getTemperature); // Initial temperature read faster
  timer.setTimeout(2000L, calculateBMI);

  // Update LCD with Blynk connection status
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Blynk Connected!");
  lcd.setCursor(0,1);
  lcd.print("Getting Data...");

  timer.setTimeout(4000L, [](){ lcd.clear(); });
}

// --- Loop Function: Runs repeatedly after setup() ---
void loop() {
  Blynk.run();
  timer.run();
}
```

Secrets.h file include the template id ,name and authorization key

```
// secrets.h
#define BLYNK_TEMPLATE_ID   "TMPL3OG0Q36oT"
#define BLYNK_TEMPLATE_NAME "project 2"
#define BLYNK_AUTH_TOKEN    "it8Dx2_WTyhBTGMKAFwMcXTaiNKQV66C"
#define WIFI_SSID           "Wokwi-GUEST"
#define WIFI_PASS           ""
```

The library of this program includes packages :


Project Libraries

Installed Libraries

DallasTemperature

OneWire

Blynk

LiquidCrystal I2C

Project link of the wokwi online simulator:
https://wokwi.com/projects/445261835299588097

Blynk cloud platform link:
https://blynk.cloud/dashboard/565773/global/devices/1/organization/565773/devices/1793461/dashboard

CONCLUSION

This ESP32-based IoT health monitoring project conclusively demonstrates that low-cost, open hardware and a cloud-connected application layer can deliver continuous vital-sign tracking, real-time visualization, and timely alerts suitable for home and community care, not just lab demos. By integrating common biomedical sensors with ESP32 and a Blynk dashboard, the system closes gaps inherent in intermittent, manual checks—turning raw readings into actionable notifications that support earlier intervention and better day-to-day self-management. The resulting prototype validates a layered architecture—sensing, edge processing, secure transport, and application/UI—that scales from a single user to multi-patient oversight while remaining accessible for students, clinicians, and makers.

The project achieves stable acquisition of temperature and user-driven or sensor-derived metrics (e.g., BMI, HR, SpO_2) on ESP32, and renders them on web/mobile dashboards with threshold-based alerts, matching established patterns from prior ESP32+Blynk implementations. Core requirements—device provisioning, datastream mapping, and real-time updates—align with best practices for virtual pins, gauges, and value displays, enabling an intuitive interface and low-latency feedback to the user or caregiver. In the broader evidence base, such IoT pipelines are associated with improved monitoring coverage and responsiveness, especially for elderly, chronic, or post-operative populations who benefit from remote observation and push notifications when values exceed safe ranges.

The working system empowers individuals to track temperature and related vitals continuously, view trends, and receive alerts without specialized hardware, reinforcing self-care and caregiver oversight between clinic visits. For educators and researchers, it provides a modular reference that can be expanded with additional sensors, predictive analytics, or security hardening, supporting coursework, capstones, or pilot deployments in community settings at modest cost. At a systems level, this matches the direction highlighted in recent IoT healthcare reviews: decentralizing monitoring, reducing manual workload, and enabling proactive responses through always-on telemetry and simple, rules-based triage.

While effective for remote observation and alerting, performance can be affected by motion artifacts (for PPG/SpO_2), Wi-Fi instability, and power constraints for long-term battery use, which can introduce missed updates or false positives if not mitigated with filtering, reconnection logic, and careful sampling intervals. The system is not a replacement for clinical diagnostics; rather, it is an adjunct that provides situational awareness and prompts timely follow-up when thresholds are breached, in line with typical claims and scope of ESP32+Blynk health prototypes. Data protection and privacy require attention beyond the core prototype—credential management, transport security, and least-privilege access—before any real patient deployment, echoing governance considerations in healthcare IoT literature.

Future directions

- Sensor and algorithm upgrades: integrate MAX3010x for HR/SpO_2 with motion-robust filtering and confidence metrics; add ambient context (BME280) and multi-sensor fusion to reduce false alerts and improve interpretability of readings.
- Resilience and scale: implement local buffering with back-pressure and reconnection backoff, add historical charts and anomaly detection in the app, and support multi-patient topics or device groups for caregiver workflows.
- Security and compliance: adopt encrypted transport, token rotation, and role-based access; design for audit logs and consent, aligning with best practices before clinical trials or institutional pilots.

In conclusion, the project substantiates that an ESP32 plus a Blynk-style cloud dashboard can deliver dependable, near-real-time health telemetry and alerts at a fraction of traditional device costs—making continuous monitoring more reachable for homes, schools, and community clinics. The design stands on a well-supported architectural pattern recognized in contemporary research and practice and is ready for targeted extensions in accuracy, resilience, and security to progress from prototype to pilot deployments. As healthcare continues to shift toward connected, preventive models, systems like this one exemplify how accessible IoT stacks can strengthen early detection, patient engagement, and the timeliness of care without imposing undue complexity or expense.