**Firmware Guide: ESP32-H2 + ADS1293**

This guide covers:

1. **Hardware Connections**
2. **ESP32-H2 Configuration**
3. **ADS1293 Initialization & Data Acquisition**
4. **BLE ECG Data Transmission**

**1. Hardware Connections**

**ESP32-H2 <-> ADS1293 (SPI)**

| **ADS1293 Pin** | **ESP32-H2 Pin** |
| --- | --- |
| VDD (3.3V) | 3.3V |
| GND | GND |
| SCLK | GPIO10 (SPI CLK) |
| MISO | GPIO11 (SPI MISO) |
| MOSI | GPIO12 (SPI MOSI) |
| CS (Chip Select) | GPIO9 |
| DRDY (Data Ready) | GPIO8 (Interrupt) |
| RST (Reset) | GPIO7 |
| PWDN (Power Down) | GND (for normal operation) |

**2. ESP32-H2 Configuration**

**Install ESP-IDF (ESP32 SDK)**

If you haven't set up ESP-IDF:

sh

CopyEdit

git clone --recursive https://github.com/espressif/esp-idf.git

cd esp-idf

./install.sh

. ./export.sh

**Create a New ESP32-H2 Project**

sh

CopyEdit

idf.py create-project esp32h2\_ecg

cd esp32h2\_ecg

**3. Initialize ADS1293 in ESP32-H2**

**SPI Driver Configuration**

Add this to main.c:

c

CopyEdit

#include <stdio.h>

#include "driver/spi\_master.h"

#include "driver/gpio.h"

#include "esp\_log.h"

#define SPI\_HOST HSPI\_HOST

#define PIN\_NUM\_MISO 11

#define PIN\_NUM\_MOSI 12

#define PIN\_NUM\_CLK 10

#define PIN\_NUM\_CS 9

#define PIN\_NUM\_DRDY 8

static const char \*TAG = "ADS1293";

spi\_device\_handle\_t ads1293;

void spi\_init() {

spi\_bus\_config\_t buscfg = {

.mosi\_io\_num = PIN\_NUM\_MOSI,

.miso\_io\_num = PIN\_NUM\_MISO,

.sclk\_io\_num = PIN\_NUM\_CLK,

.quadwp\_io\_num = -1,

.quadhd\_io\_num = -1,

};

spi\_device\_interface\_config\_t devcfg = {

.clock\_speed\_hz = 1 \* 1000 \* 1000, // 1 MHz

.mode = 0,

.spics\_io\_num = PIN\_NUM\_CS,

.queue\_size = 1,

};

spi\_bus\_initialize(SPI\_HOST, &buscfg, SPI\_DMA\_CH\_AUTO);

spi\_bus\_add\_device(SPI\_HOST, &devcfg, &ads1293);

ESP\_LOGI(TAG, "SPI Initialized");

}

uint8\_t spi\_read\_register(uint8\_t reg) {

uint8\_t tx\_data[2] = {reg | 0x80, 0}; // Read command

uint8\_t rx\_data[2];

spi\_transaction\_t t = {

.length = 16,

.tx\_buffer = tx\_data,

.rx\_buffer = rx\_data,

};

spi\_device\_transmit(ads1293, &t);

return rx\_data[1];

}

void spi\_write\_register(uint8\_t reg, uint8\_t data) {

uint8\_t tx\_data[2] = {reg, data};

spi\_transaction\_t t = {

.length = 16,

.tx\_buffer = tx\_data,

.rx\_buffer = NULL,

};

spi\_device\_transmit(ads1293, &t);

}

**4. Configure ADS1293 for ECG Acquisition**

Add this to main.c:

c

CopyEdit

void ads1293\_init() {

spi\_write\_register(0x00, 0x01); // Reset ADS1293

spi\_write\_register(0x01, 0x03); // Enable channels

spi\_write\_register(0x02, 0x10); // Set sampling rate

spi\_write\_register(0x03, 0x07); // ECG gain settings

spi\_write\_register(0x04, 0x01); // Enable lead-off detection

ESP\_LOGI(TAG, "ADS1293 Initialized");

}

void ads1293\_read\_data() {

uint8\_t ecg\_data[3];

ecg\_data[0] = spi\_read\_register(0x20); // ECG channel 1

ecg\_data[1] = spi\_read\_register(0x21); // ECG channel 2

ecg\_data[2] = spi\_read\_register(0x22); // ECG channel 3

ESP\_LOGI(TAG, "ECG Data: %02X %02X %02X", ecg\_data[0], ecg\_data[1], ecg\_data[2]);

}

**5. BLE Data Transmission (ECG to Mobile)**

**Enable BLE in sdkconfig**

Run:

sh

CopyEdit

idf.py menuconfig

Enable:

* **Component Config -> Bluetooth -> Enable Bluetooth**
* **Component Config -> Bluetooth -> BLE**

**BLE Service (GATT) for ECG Data**

Add this to ble\_server.c:

c

CopyEdit

#include "esp\_log.h"

#include "esp\_bt.h"

#include "esp\_gatt\_defs.h"

#include "esp\_gatts\_api.h"

#include "esp\_gap\_ble\_api.h"

#include "esp\_bt\_main.h"

static const char \*TAG = "BLE\_GATT";

// ECG UUIDs

#define ECG\_SERVICE\_UUID 0x180D

#define ECG\_CHARACTERISTIC\_UUID 0x2A37

void ble\_init() {

esp\_bt\_controller\_mem\_release(ESP\_BT\_MODE\_CLASSIC\_BT);

esp\_bt\_controller\_init();

esp\_bt\_controller\_enable(ESP\_BT\_MODE\_BLE);

ESP\_LOGI(TAG, "BLE Initialized");

}

void send\_ecg\_data(uint8\_t \*data, size\_t length) {

esp\_ble\_gatts\_send\_indicate(/\* gatt\_if \*/, /\* conn\_id \*/, /\* char\_handle \*/, length, data, false);

}

**6. Main Execution**

Modify main.c:

c

CopyEdit

void app\_main() {

spi\_init();

ads1293\_init();

ble\_init();

while (1) {

ads1293\_read\_data();

vTaskDelay(pdMS\_TO\_TICKS(100)); // 100ms delay

}

}

**7. Mobile App for ECG Visualization**

For an **Android/iOS BLE App**:

* **Android (Kotlin) BLE Library:** [Android BLE Scanner](https://developer.android.com/guide/topics/connectivity/bluetooth)
* **Flutter BLE Library:** FlutterBlue

**8. Power Optimization for Wearable**

* **Deep Sleep Mode for ESP32-H2:**

c

CopyEdit

esp\_sleep\_enable\_timer\_wakeup(5000000); // 5 seconds

esp\_deep\_sleep\_start();

* **ADS1293 Low-Power Mode:**

c

CopyEdit

spi\_write\_register(0x01, 0x00); // Power-down mode

**Final Notes**

🔹 **ESP32-H2 + ADS1293 is a cost-effective ECG solution!**  
🔹 **BLE 5.3 ensures stable ECG data transmission.**  
🔹 **Further steps: Filter ECG signals, optimize power, and integrate a mobile app.**

Would you like an **ECG filtering algorithm** or **PCB design recommendations** next? 🚀

4o

**You said:**

can you provide arduino code?

**ChatGPT said:**

Yes! Here's the **Arduino code** for using **ESP32-H2** with **ADS1293** and sending ECG data via BLE.

**📌 Features**

✅ **SPI Communication** with ADS1293  
✅ **Real-time ECG Data Acquisition**  
✅ **BLE Data Transmission** to a mobile app  
✅ **Optimized Power Management**

**🔌 Wiring: ESP32-H2 & ADS1293**

| **ADS1293 Pin** | **ESP32-H2 Pin** |
| --- | --- |
| VDD (3.3V) | 3.3V |
| GND | GND |
| SCLK | GPIO10 (SPI CLK) |
| MISO | GPIO11 (SPI MISO) |
| MOSI | GPIO12 (SPI MOSI) |
| CS (Chip Select) | GPIO9 |
| DRDY (Data Ready) | GPIO8 |
| RST (Reset) | GPIO7 |
| PWDN (Power Down) | GND |

**🚀 Arduino Code for ESP32-H2**

cpp

CopyEdit

#include <Arduino.h>

#include <SPI.h>

#include <BLEDevice.h>

#include <BLEUtils.h>

#include <BLEServer.h>

#define SPI\_MISO 11

#define SPI\_MOSI 12

#define SPI\_SCLK 10

#define SPI\_CS 9

#define DRDY\_PIN 8

BLEServer \*pServer = NULL;

BLECharacteristic \*ecgCharacteristic;

bool deviceConnected = false;

#define ECG\_SERVICE\_UUID "180D"

#define ECG\_CHARACTERISTIC\_UUID "2A37"

// BLE Callback

class MyServerCallbacks: public BLEServerCallbacks {

void onConnect(BLEServer\* pServer) {

deviceConnected = true;

};

void onDisconnect(BLEServer\* pServer) {

deviceConnected = false;

}

};

// Initialize SPI

void spi\_init() {

pinMode(SPI\_CS, OUTPUT);

digitalWrite(SPI\_CS, HIGH);

SPI.begin(SPI\_SCLK, SPI\_MISO, SPI\_MOSI, SPI\_CS);

}

// SPI Read

uint8\_t spi\_read\_register(uint8\_t reg) {

digitalWrite(SPI\_CS, LOW);

SPI.transfer(reg | 0x80); // Read command

uint8\_t data = SPI.transfer(0);

digitalWrite(SPI\_CS, HIGH);

return data;

}

// SPI Write

void spi\_write\_register(uint8\_t reg, uint8\_t data) {

digitalWrite(SPI\_CS, LOW);

SPI.transfer(reg);

SPI.transfer(data);

digitalWrite(SPI\_CS, HIGH);

}

// Initialize ADS1293

void ads1293\_init() {

spi\_write\_register(0x00, 0x01); // Reset ADS1293

delay(100);

spi\_write\_register(0x01, 0x03); // Enable ECG channels

spi\_write\_register(0x02, 0x10); // Set sample rate

spi\_write\_register(0x03, 0x07); // Gain settings

spi\_write\_register(0x04, 0x01); // Enable lead-off detection

}

// Read ECG Data

void ads1293\_read\_data(uint8\_t \*ecg\_data) {

ecg\_data[0] = spi\_read\_register(0x20); // ECG channel 1

ecg\_data[1] = spi\_read\_register(0x21); // ECG channel 2

ecg\_data[2] = spi\_read\_register(0x22); // ECG channel 3

}

// Initialize BLE

void ble\_init() {

BLEDevice::init("ESP32H2\_ECG");

pServer = BLEDevice::createServer();

pServer->setCallbacks(new MyServerCallbacks());

BLEService \*pService = pServer->createService(ECG\_SERVICE\_UUID);

ecgCharacteristic = pService->createCharacteristic(

ECG\_CHARACTERISTIC\_UUID,

BLECharacteristic::PROPERTY\_READ |

BLECharacteristic::PROPERTY\_NOTIFY

);

pService->start();

BLEAdvertising \*pAdvertising = BLEDevice::getAdvertising();

pAdvertising->addServiceUUID(ECG\_SERVICE\_UUID);

pAdvertising->start();

}

// Main Setup

void setup() {

Serial.begin(115200);

spi\_init();

ads1293\_init();

ble\_init();

pinMode(DRDY\_PIN, INPUT);

}

// Main Loop

void loop() {

if (digitalRead(DRDY\_PIN) == LOW) { // New ECG data ready

uint8\_t ecg\_data[3];

ads1293\_read\_data(ecg\_data);

Serial.printf("ECG: %02X %02X %02X\n", ecg\_data[0], ecg\_data[1], ecg\_data[2]);

if (deviceConnected) {

ecgCharacteristic->setValue(ecg\_data, 3);

ecgCharacteristic->notify();

}

}

delay(10); // Small delay to reduce CPU usage

}

**📱 Mobile App for ECG Display**

To receive ECG data on your phone, use:

* **nRF Connect App** (iOS/Android)
* **Flutter Blue / Android BLE API** for a custom app

**🔋 Power Optimization**

* Enable **deep sleep**:

cpp

CopyEdit

esp\_sleep\_enable\_timer\_wakeup(5000000); // Sleep for 5 seconds

esp\_deep\_sleep\_start();

* Reduce **ADS1293 power consumption**:

cpp

CopyEdit

spi\_write\_register(0x01, 0x00); // Put ADS1293 in low-power mode

**🎯 Next Steps**

Would you like:

1. **ECG Signal Filtering (Low-pass, High-pass, Notch)?**
2. **PCB Design Recommendations (Battery, Electrodes, Enclosure)?**
3. **Mobile App Development Guide for Real-time ECG Display?**

Let me know, and I'll guide you through the next phase! 🚀

**You said:**

Yes, ECG signal filtering

**ChatGPT said:**

**📌 ECG Signal Filtering for ESP32-H2 & ADS1293**

Raw ECG signals often contain **noise** from muscle activity (EMG), power lines (50/60Hz), and baseline drift. To improve signal quality, we apply **digital filtering**.

**🔹 ECG Noise Sources & Solutions**

| **Noise Type** | **Frequency Range** | **Solution** |
| --- | --- | --- |
| **Baseline Drift** | < 0.5 Hz | **High-Pass Filter (0.5 Hz cutoff)** |
| **Power Line Noise** | 50/60 Hz | **Notch Filter (50/60 Hz removal)** |
| **EMG (Muscle Noise)** | 20-500 Hz | **Low-Pass Filter (40 Hz cutoff)** |

**🔹 Digital Filtering Using DSP**

We apply **Finite Impulse Response (FIR) & Infinite Impulse Response (IIR)** filters.

**✅ Step 1: Implement High-Pass Filter (HPF)**

Removes **baseline drift** (frequencies below 0.5 Hz).

cpp

CopyEdit

#define HPF\_ALPHA 0.99 // Smoothing factor (0.9 - 0.99 recommended)

float previousECG = 0;

float highPassFilter(float input) {

float output = HPF\_ALPHA \* (previousECG + input - previousECG);

previousECG = output;

return output;

}

**✅ Step 2: Implement Low-Pass Filter (LPF)**

Removes **muscle noise** (frequencies above 40 Hz).

cpp

CopyEdit

#define LPF\_ALPHA 0.1 // Smoothing factor

float previousLPF = 0;

float lowPassFilter(float input) {

float output = LPF\_ALPHA \* input + (1 - LPF\_ALPHA) \* previousLPF;

previousLPF = output;

return output;

}

**✅ Step 3: Implement Notch Filter (50/60 Hz)**

Removes **power line noise**.

cpp

CopyEdit

#define NOTCH\_FREQ 50 // Set to 60 if using 60Hz power supply

#define SAMPLE\_RATE 200 // ECG sampling rate in Hz

float notchFilter(float input) {

static float y[3] = {0};

static float x[3] = {0};

// Notch filter coefficients (for 50Hz at 200Hz sample rate)

float b0 = 0.9726, b1 = -1.9070, b2 = 0.9726;

float a1 = -1.9070, a2 = 0.9453;

x[0] = input;

y[0] = b0\*x[0] + b1\*x[1] + b2\*x[2] - a1\*y[1] - a2\*y[2];

x[2] = x[1];

x[1] = x[0];

y[2] = y[1];

y[1] = y[0];

return y[0];

}

**🔹 Complete ECG Processing Code**

Now, we integrate all filters into your **BLE ECG system**.

cpp

CopyEdit

void ads1293\_read\_data(uint8\_t \*ecg\_data) {

ecg\_data[0] = spi\_read\_register(0x20);

ecg\_data[1] = spi\_read\_register(0x21);

ecg\_data[2] = spi\_read\_register(0x22);

// Convert raw data to float ECG value

float rawECG = (ecg\_data[0] << 16 | ecg\_data[1] << 8 | ecg\_data[2]) / 1000000.0;

// Apply filters

float filteredECG = highPassFilter(rawECG);

filteredECG = lowPassFilter(filteredECG);

filteredECG = notchFilter(filteredECG);

Serial.printf("ECG Filtered: %f\n", filteredECG);

}

**🔹 Next Steps**

Would you like:

1. **ECG Signal Visualization (Matplotlib / Serial Plotter)?**
2. **Adaptive Noise Cancellation (Machine Learning-Based)?**
3. **PCB Design Guide (Electrode Placement & Grounding)?**

Let me know, and I'll guide you through the next steps! 🚀

**You said:**

Give me 2 and 3 step

**ChatGPT said:**

**🔹 Next Steps for ECG Signal Processing**

Now, we'll cover **Adaptive Noise Cancellation (ANC) using Machine Learning** and **PCB Design Best Practices for ECG Electrodes and Grounding**.

**Step 2: Adaptive Noise Cancellation (ML-Based)**

Instead of static filters, we use **Machine Learning (ML) to adaptively remove noise**.  
We will use a **Neural Network (LSTM)** to distinguish between noise and real ECG signals.

**🔹 Why Use ML for Noise Cancellation?**

Traditional filters remove **fixed-frequency noise**, but ECG noise is **dynamic** (e.g., movement artifacts).  
ML can **learn** to remove unwanted noise in real time.

**✅ Solution: Use an LSTM-Based Model**

* **LSTM (Long Short-Term Memory)** learns **temporal dependencies** in ECG data.
* It can distinguish between **real heartbeats** and **artifacts**.

**🔹 Implementation Steps**

**1️⃣ Collect Raw ECG Data (With and Without Noise)**

Save **raw ECG signals** to a CSV file from your ESP32-H2:

cpp

CopyEdit

void logECGData(float ecg\_value, float noisy\_value) {

Serial.printf("%f,%f\n", ecg\_value, noisy\_value); // Save in CSV format

}

📌 **Next Step:** Collect ECG data while moving (with artifacts) and when still.

**2️⃣ Train an LSTM Model in Python (Using TensorFlow)**

Once you have collected ECG data, train a **LSTM-based Denoiser**:

**Install TensorFlow**

sh

CopyEdit

pip install tensorflow numpy pandas matplotlib

**Train the Model**

python

CopyEdit

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

# Load ECG Data

data = pd.read\_csv("ecg\_data.csv")

X\_train = np.array(data["noisy\_ecg"]).reshape(-1, 1, 1) # Input: Noisy ECG

y\_train = np.array(data["clean\_ecg"]).reshape(-1, 1, 1) # Output: Clean ECG

# Build LSTM Model

model = Sequential([

LSTM(50, return\_sequences=True, input\_shape=(1, 1)),

LSTM(50, return\_sequences=True),

Dense(1) # Output: Clean ECG Signal

])

model.compile(loss='mse', optimizer='adam')

model.fit(X\_train, y\_train, epochs=20, batch\_size=32)

# Save Model

model.save("ecg\_denoiser.h5")

**3️⃣ Deploy Model on ESP32-H2**

Convert the trained model to **TensorFlow Lite (TFLite)** and run it on ESP32-H2.

**Convert Model to TFLite**

python

CopyEdit

converter = tf.lite.TFLiteConverter.from\_saved\_model("ecg\_denoiser.h5")

tflite\_model = converter.convert()

with open("ecg\_denoiser.tflite", "wb") as f:

f.write(tflite\_model)

**Run TFLite Model on ESP32-H2**

cpp

CopyEdit

#include <TensorFlowLite.h>

tflite::Model\* model;

tflite::MicroInterpreter\* interpreter;

void setup() {

model = tflite::GetModel(ecg\_denoiser\_tflite);

interpreter = new tflite::MicroInterpreter(model, tensor\_arena, tensor\_arena\_size);

interpreter->AllocateTensors();

}

float denoiseECG(float noisy\_signal) {

interpreter->input(0)->data.f[0] = noisy\_signal;

interpreter->Invoke();

return interpreter->output(0)->data.f[0];

}

📌 **Now, ESP32-H2 can remove ECG noise in real-time!** 🚀

**Step 3: PCB Design Guide for ECG Electrodes & Grounding**

To get a **clean ECG signal**, we need proper **PCB design & electrode placement**.

**🔹 Key PCB Design Considerations**

| **Design Aspect** | **Recommendation** |
| --- | --- |
| **Electrode Type** | Silver/Silver Chloride (Ag/AgCl) |
| **ADC Reference Voltage** | 3.3V for ESP32-H2 |
| **PCB Layer** | **4-layer PCB** (Power, GND, Signals, Shield) |
| **Grounding** | Analog & Digital GND separated |
| **Shielding** | Use a copper ground plane under ECG traces |
| **Isolation** | Use **Optocouplers** to isolate BLE & ECG |
| **Power Source** | LiPo Battery + Low-Noise LDO |

**🔹 Circuit Schematic for ECG Front-End**

**🛠 Components:**

* **AFE (Analog Front-End):** **ADS1293**
* **MCU:** **ESP32-H2**
* **Op-Amps:** **INA333 (Instrumentation Amp)**
* **Electrodes:** **Ag/AgCl ECG Pads**
* **Power Supply:** **3.3V LDO (LP5907)**

**Basic ECG Acquisition Circuit**

css

CopyEdit

[Electrode 1] ----> [INA333] ---> [ADS1293] ---> [ESP32-H2]

[Electrode 2] ----> [INA333] ---> [ADS1293] ---> [ESP32-H2]

[Reference Electrode] ---> [GND]

**🔹 PCB Layout Guidelines**

✅ **Keep ECG Traces Short:**

* Minimize trace length between **electrodes & AFE**.
* Use **differential pairs** for ECG signals.

✅ **Use a Dedicated Ground Plane:**

* Separate **analog & digital ground (AGND, DGND)**.
* Connect both at a **single point (star grounding)**.

✅ **Shield ECG Traces:**

* Use **a ground plane under ECG signals** to reduce noise.
* **No high-frequency signals** near ECG traces.

✅ **Place ADS1293 Close to Electrodes:**

* Reduce input impedance & improve signal quality.

✅ **Use Right-Leg Drive (RLD) for Noise Cancellation:**

* Connect a **right-leg drive circuit** to improve ECG signal quality.

**🔹 Recommended PCB Stack-Up**

| **Layer** | **Purpose** |
| --- | --- |
| **Top Layer** | ECG Signals, AFE, MCU |
| **2nd Layer** | Solid Ground Plane |
| **3rd Layer** | Power Distribution |
| **Bottom Layer** | BLE Antenna & Shielding |

**🚀 Next Steps**

Would you like:

1. **A Full Schematic & PCB Design (Altium / KiCad)?**
2. **A Mobile App (Flutter / Android) to Display ECG?**
3. **Battery Management System (BMS) for Wearable ECG?**

Let me know your priority, and I'll guide you further! 🚀

**You said:**

Yes, 1 and 3

**ChatGPT said:**

Great! Let's dive into **1. Full Schematic & PCB Design** and **2. Battery Management System (BMS)** for your wearable Holter monitor.

**1️⃣ Full Schematic & PCB Design for ECG Monitor**

**🛠 Key Components Overview:**

1. **ECG Signal Acquisition:**
   * **Electrodes:** Ag/AgCl ECG Pads (3 or 5 leads depending on design).
   * **Instrumentation Amplifier:** **INA333** or **INA126** for ECG signal conditioning.
   * **AFE:** **ADS1293** for ADC conversion, gain control, and filtering.
2. **Microcontroller:**
   * **ESP32-H2**: Central MCU for BLE communication, sensor data processing, and noise filtering.
3. **Power Supply:**
   * **LDO Regulator**: **LP5907** for a stable 3.3V supply to the ESP32-H2 and ECG sensor.
   * **Li-ion Battery**: For portable power.
   * **Battery Charging Circuit**: Includes **TP4056** charging module with protection.
4. **BLE Communication:**
   * The **ESP32-H2** will send filtered ECG data to the mobile phone via BLE.

**🛠 Circuit Schematic:**

The following describes how to connect the components:

1. **ECG Front-End (INA333, ADS1293)**
   * **INA333** takes the differential signals from the electrodes and amplifies them.
   * The output of the **INA333** is sent to the **ADS1293**, which acts as the **Analog-to-Digital Converter** (ADC) and performs digital filtering.
   * The **ADS1293** will interface with the **ESP32-H2** via SPI to send the digitized ECG signal.
2. **Power Supply (LDO & Li-ion Battery)**
   * The **LP5907 LDO** will provide a clean 3.3V to power the **ESP32-H2** and other low-power components.
   * The **TP4056** module will charge a **Li-ion Battery** (e.g., 3.7V 2000mAh).

**PCB Design Considerations:**

1. **Layer Stack:**
   * **Top Layer:** Placement for ECG traces, AFE (INA333, ADS1293), ESP32-H2, and power.
   * **Ground Plane:** Use a solid ground plane for both **Analog (AGND)** and **Digital (DGND)**, connected at a single point.
   * **Bottom Layer:** **BLE Antenna**, and shielding for the analog circuitry.
2. **Component Placement:**
   * **Electrode Inputs:** Place the **electrode connections** close to the **INA333** to minimize noise.
   * **ADS1293 & ESP32-H2** should be placed close to minimize trace lengths, especially for SPI signals.
   * **Battery and Power Management:** Ensure **Li-ion Battery** is safely placed with a protection circuit, keeping it away from high-power components.

**🛠 Recommended Tools for PCB Design:**

1. **Altium Designer** or **KiCad** (for schematic capture and PCB layout).
2. **Libraries:**
   * Use the **ADS1293**, **INA333**, and **ESP32-H2** libraries for footprint and symbol creation in your design tool.

**2️⃣ Battery Management System (BMS) for Wearable ECG**

For your **wearable Holter monitor**, **battery management** is critical to ensure safe and reliable operation. Here's how you can design the BMS for your device:

**🛠 Components Needed for BMS:**

1. **Li-ion Battery (3.7V)**
2. **TP4056 Charging Module**: Simple charging and protection for your battery.
3. **Battery Protection Circuit (IC)**: Ensures overcharge, over-discharge, and overcurrent protection.
   * A suitable IC could be **DW01** or **BMS protection IC** for Li-ion cells.
4. **Low Dropout Regulator (LDO)**: **LP5907** to provide stable 3.3V to the system.

**🛠 How the BMS Works:**

1. **Battery Charging (TP4056 Module)**:
   * The **TP4056** charging module will safely charge your **Li-ion battery**. It also includes overcharge protection, overcurrent protection, and automatic switching to charging mode when the battery voltage drops.
2. **Battery Protection (DW01 or BMS IC)**:
   * Use a **DW01** protection IC in combination with **FETs** to provide battery protection against:
     + **Overcharge** (stopping charging when the voltage exceeds the maximum threshold, e.g., 4.2V).
     + **Over-discharge** (disconnecting the load when the voltage drops below the minimum threshold, e.g., 3.0V).
     + **Overcurrent** (disconnecting the load if excessive current is drawn).
3. **Power Distribution (LDO Regulator)**:
   * The **LP5907 LDO** will convert the **Li-ion battery voltage (3.7V)** to the required **3.3V** supply for the **ESP32-H2**, **ADS1293**, and other components.

**🛠 Example Schematic for BMS:**

plaintext

CopyEdit

[Li-ion Battery] ---> [TP4056 Charging Module] ---> [DW01 Protection IC] ---> [ESP32-H2, ADS1293, etc.]

(Charge) (Protection) (3.3V Power)

1. The **TP4056** module manages charging and supplies power when the battery is not charging.
2. The **DW01 Protection IC** ensures that the battery is protected from overcharging and overdischarge.
3. The **LP5907 LDO** provides a stable 3.3V supply for your system.

**🛠 PCB Layout for BMS:**

1. **Li-ion Battery:** Place the battery near the power circuit but ensure it has proper thermal dissipation to prevent overheating.
2. **Charging Circuit (TP4056):** Place the charging module away from high-noise components.
3. **Protection Circuit (DW01):** Ensure it is connected properly to the battery and load to manage current flow safely.

**Recommended BMS Components:**

1. **TP4056**: Charging IC for Li-ion batteries.
2. **DW01**: Battery protection IC.
3. **LP5907**: Low Dropout Regulator (LDO) for 3.3V.