Hardware Code

```
#include <Arduino.h>
#include <bluefruit.h>
#include <movingAvg.h>
#include <arduinoFFT.h>
#include <Adafruit_LSM6DS3.h> // For the onboard IMU

// BLE Configuration
BLEService healthService("1810");
BLECharacteristic hrChar("2A37", BLERead | BLENotify, 2);
BLECharacteristic spO2Char("2A5F", BLERead | BLENotify, 2);
BLECharacteristic eegChar("2A38", BLERead | BLENotify, 20); // EEG data
packet
BLECharacteristic battChar("2A19", BLERead | BLENotify, 1);

// Pins
#define VIBRATION_PIN D2
#define BATTERY_PIN A0
#define EEG_INPUT_PIN A1  // BioAmp EXG Pill output

// EEG Configuration
const uint16_t EEG_SAMPLES = 128; // Must be power of 2
const float SAMPLING_RATE = 250.0; // Hz
ArduinoFFT<double> FFT;
double vReal[EEG_SAMPLES];
double vImag[EEG_SAMPLES];
unsigned long samplingPeriod;

// Moving averages
movingAvg hrAvg(5);
movingAvg spO2Avg(5);
movingAvg eegAvg(5);

// Battery params
```

```
const float BAT_MAX = 3.7f;
const float BAT_MIN = 3.0f;

// EEG Band Definitions (Hz)
enum EEGBand {
  DELTA = 0, // 0.5-4Hz
  THETA,     // 4-8Hz
  ALPHA,     // 8-12Hz
  BETA,      // 12-30Hz
  GAMMA,     // 30-100Hz
  BAND_COUNT
};

enum EmergencyType {
  HR_EMERGENCY,
  SPO2_EMERGENCY,
  SPO2_CRITICAL,
  EEG_ARTIFACT_DETECTED,
  SEIZURE_PATTERN,
  FALL_DETECTED
};



// Struct for EEG analysis
struct EEGAnalysis {
  uint8_t bandPercents[BAND_COUNT];
  uint8_t dominantBand;
};

Adafruit_LSM6DS3 lsm6ds3; // IMU object
// Fall detection parameters
const float FALL_ACCEL_THRESHOLD = 2.5f; // g-force threshold (2.5g)
const float FREE_FALL_THRESHOLD = 0.5f;   // g-force threshold for free-fa
ll
const unsigned long IMPACT_DURATION = 1000; // ms to detect post-impa
ct stillness
const float POST_FALL_ANGLE = 45.0f;      // degrees from vertical
```

```
// Variables for fall detection
unsigned long fallStartTime = 0;
bool potentialFall = false;
float preFallOrientation[3] = {0};

void setup() {
  Serial.begin(115200);
  while(!Serial) delay(10);

  // Initialize hardware
  pinMode(VIBRATION_PIN, OUTPUT);
  pinMode(EEG_INPUT_PIN, INPUT);

  hrAvg.begin();
  spO2Avg.begin();
  eegAvg.begin();

  // Calculate sampling period
  samplingPeriod = 1000000 / SAMPLING_RATE;

  // Initialize BLE
  Bluefruit.begin();
  Bluefruit.setName("MindHealth_Tracker");
  Bluefruit.setTxPower(4);

  setupBLE();
  Serial.println("BLE Ready");
}

void setupBLE() {
  healthService.begin();

  hrChar.setProperties(BLERead | BLENotify);
  hrChar.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrChar.begin();

  spO2Char.setProperties(BLERead | BLENotify);
  spO2Char.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
```

```
  spO2Char.begin();

  eegChar.setProperties(BLERead | BLENotify);
  eegChar.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  eegChar.begin();

  battChar.setProperties(BLERead | BLENotify);
  battChar.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  battChar.begin();

  Bluefruit.Advertising.addService(healthService);
  Bluefruit.Advertising.start();
}

void loop() {
  static uint32_t lastSampleTime = 0;
  static uint16_t sampleCount = 0;

  // // BLE maintenance
  // Bluefruit.process();

  // EEG Sampling
  if (micros() - lastSampleTime >= samplingPeriod && sampleCount < EEG_
SAMPLES) {
    vReal[sampleCount] = analogRead(EEG_INPUT_PIN);
    vImag[sampleCount] = 0;
    sampleCount++;
    lastSampleTime = micros();
  }

  // Process EEG when buffer full
  if (sampleCount >= EEG_SAMPLES) {
    EEGAnalysis analysis = processEEG();
    sendEEGData(analysis);
    sampleCount = 0;
  }

  // Simulate HR/SpO2 (replace with actual MAX30102 code)
```

```
  if (millis() % 1000 == 0) {
    updateHRSpO2();
    checkBattery();
  }

  checkEmergencies();
}

EEGAnalysis processEEG() {
  EEGAnalysis result = {{0}, 0};

  // Apply FFT
  FFT.windowing(vReal, EEG_SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FO
RWARD);
  FFT.compute(vReal, vImag, EEG_SAMPLES, FFT_FORWARD);
  FFT.complexToMagnitude(vReal, vImag, EEG_SAMPLES);

  // Classify frequency bands
  float bandTotals[BAND_COUNT] = {0};
  float totalPower = 0;

  for (int i = 1; i < EEG_SAMPLES/2; i++) {
    float freq = i * (SAMPLING_RATE / EEG_SAMPLES);
    float magnitude = vReal[i];

    if (freq < 4) bandTotals[DELTA] += magnitude;
    else if (freq < 8) bandTotals[THETA] += magnitude;
    else if (freq < 12) bandTotals[ALPHA] += magnitude;
    else if (freq < 30) bandTotals[BETA] += magnitude;
    else if (freq < 100) bandTotals[GAMMA] += magnitude;

    totalPower += magnitude;
  }

  // Calculate percentages
  if (totalPower > 0) {
    for (int i = 0; i < BAND_COUNT; i++) {
      result.bandPercents[i] = (bandTotals[i] / totalPower) * 100;
```

```
    }

    // Find dominant band
    float maxVal = 0;
    for (int i = 0; i < BAND_COUNT; i++) {
      if (bandTotals[i] > maxVal) {
        maxVal = bandTotals[i];
        result.dominantBand = i;
      }
    }
  }

  return result;
}

void sendEEGData(EEGAnalysis analysis) {
  uint8_t eegData[20] = {0};

  // Format: [dominantBand, delta%, theta%, alpha%, beta%, gamma%]
  eegData[0] = analysis.dominantBand;
  for (int i = 0; i < BAND_COUNT; i++) {
    eegData[i+1] = analysis.bandPercents[i];
  }

  eegChar.write(eegData, sizeof(eegData));
}

void updateHRSpO2() {
  // Simulate data (replace with MAX30102 readings)
  uint16_t hr = 70 + random(-5, 5);
  uint16_t spo2 = 97 + random(-1, 1);

  hrAvg.reading(hr);
  spO2Avg.reading(spo2);

  uint16_t hrValue = hrAvg.getAvg();
  uint16_t spO2Value = spO2Avg.getAvg();
  hrChar.write(&hrValue, sizeof(uint16_t));
```

```
  spO2Char.write(&spO2Value, sizeof(uint16_t));

}

void checkBattery() {
  int raw = analogRead(BATTERY_PIN);
  float voltage = (raw * 3.3f / 1024.0f) * 2; // Voltage divider
  uint8_t level = map(voltage * 100, BAT_MIN * 100, BAT_MAX * 100, 0, 100);

  battChar.write(&level, 1);

  if (level < 20) {
    triggerVibration(2, 500); // Double buzz for low battery
  }
}

void checkEmergencies() {
  static uint32_t lastCheckTime = 0;
  if (millis() - lastCheckTime < 1000) return; // Run once per second
  lastCheckTime = millis();

  // Current vital readings
  float currentHR = hrAvg.getAvg();
  float currentSpO2 = spO2Avg.getAvg();

  // Threshold values (customize as needed)
  const float HR_CRITICAL_LOW = 40.0f;
  const float HR_CRITICAL_HIGH = 120.0f;
  const float SpO2_CRITICAL = 90.0f;
  const float EEG_ARTIFACT_THRESHOLD = 500.0f; // Raw EEG amplitude t
hreshold

  // Check heart rate emergencies
  if (currentHR < HR_CRITICAL_LOW || currentHR > HR_CRITICAL_HIGH) {
    triggerEmergencyAlert(HR_EMERGENCY);
    Serial.print("HR Emergency: ");
    Serial.println(currentHR);
  }
```

```
  // Check SpO2 emergencies
  if (currentSpO2 < SpO2_CRITICAL) {
    triggerEmergencyAlert(SPO2_EMERGENCY);
    Serial.print("SpO2 Emergency: ");
    Serial.println(currentSpO2);
  }

  // EEG Spike Detection (simple amplitude check)
  for (int i = 0; i < EEG_SAMPLES; i++) {
    if (abs(vReal[i]) > EEG_ARTIFACT_THRESHOLD) {
      triggerEmergencyAlert(EEG_ARTIFACT_DETECTED);
      Serial.println("EEG Artifact Detected");
      break;
    }
  }

  // Check for seizure-like patterns (high gamma dominance)
  EEGAnalysis latestAnalysis = processEEG();
  if (latestAnalysis.bandPercents[GAMMA] > 40 &&
      latestAnalysis.bandPercents[BETA] > 30) {
    triggerEmergencyAlert(SEIZURE_PATTERN);
    Serial.println("Potential seizure pattern detected");
  }

  // Fall detection using IMU (if available)
  if (isFallingDetected()) {
    triggerEmergencyAlert(FALL_DETECTED);
    Serial.println("Fall detected");
  }
}

void triggerEmergencyAlert(EmergencyType type) {
  switch(type) {
    case HR_EMERGENCY:
      // Triple vibration for heart emergency
      triggerVibration(3, 300);
      break;
```

```
    case SPO2_CRITICAL:
      // Double long vibration for SpO2
      triggerVibration(2, 500);
      break;

    case EEG_ARTIFACT_DETECTED:
      // Quick bursts for EEG issues
      triggerVibration(4, 100);
      break;

    case SEIZURE_PATTERN:
      // Continuous vibration for seizure
      digitalWrite(VIBRATION_PIN, HIGH);
      delay(3000); // 3 second continuous
      digitalWrite(VIBRATION_PIN, LOW);
      break;

    case FALL_DETECTED:
      // Alternating pattern for fall
      for (int i = 0; i < 5; i++) {
        digitalWrite(VIBRATION_PIN, HIGH);
        delay(100);
        digitalWrite(VIBRATION_PIN, LOW);
        delay(100);
      }
      break;
  }
}

void setupIMU() {
  if (!lsm6ds3.begin_I2C()) {
    Serial.println("Failed to initialize LSM6DS3 IMU");
    while (1);
  }

  // Configure IMU settings
  lsm6ds3.setAccelRange(LSM6DS_ACCEL_RANGE_16_G);
```

```
  lsm6ds3.setAccelDataRate(LSM6DS_RATE_104_HZ);
  lsm6ds3.setGyroRange(LSM6DS_GYRO_RANGE_2000_DPS);
  lsm6ds3.setGyroDataRate(LSM6DS_RATE_104_HZ);
}

bool isFallingDetected() {
  sensors_event_t accel;
  sensors_event_t gyro;
  sensors_event_t temp;
  lsm6ds3.getEvent(&accel, &gyro, &temp);
  // Calculate total acceleration magnitude
  float accelMag = sqrt(sq(accel.acceleration.x) +
                     sq(accel.acceleration.y) +
                     sq(accel.acceleration.z));
  // Detect free-fall (low g-force)
  if (accelMag < FREE_FALL_THRESHOLD && !potentialFall) {
    potentialFall = true;
    fallStartTime = millis();

    // Store pre-fall orientation
    preFallOrientation[0] = accel.acceleration.x;
    preFallOrientation[1] = accel.acceleration.y;


    // Check post-impact stillness
    if (checkPostFallStillness()) {
      // Verify orientation change
      float currentOrientation[3] = {
        accel.acceleration.x,
        accel.acceleration.y,
        accel.acceleration.z
      };

      float angleChange = calculateOrientationChange(preFallOrientation, cur
rentOrientation);

      if (angleChange > POST_FALL_ANGLE) {
        potentialFall = false;
```

```
      return true;
    }
  }
}
// Reset if too much time has passed
if (potentialFall && (millis() - fallStartTime) > 2000) {
  potentialFall = false;
}
return false;
}


bool checkPostFallStillness() {
  unsigned long startTime = millis();
  float initialAccel[3];

  sensors_event_t accel;
  lsm6ds3.getEvent(&accel, NULL, NULL);

  initialAccel[1] = accel.acceleration.y;
  initialAccel[2] = accel.acceleration.z;

  while (millis() - startTime < IMPACT_DURATION) {
    lsm6ds3.getEvent(&accel, NULL, NULL);

    // Check for significant movement
    float diff = sqrt(sq(accel.acceleration.x - initialAccel[0]) +
                 sq(accel.acceleration.y - initialAccel[1]) +
                 sq(accel.acceleration.z - initialAccel[2]));

    if (diff > 0.3f) { // Movement threshold
      return false;
    }

    delay(50);
  }

  return true;
```

```
}

float calculateOrientationChange(float before[3], float after[3]) {
  // Calculate dot product
  float dot = before[0]*after[0] + before[1]*after[1] + before[2]*after[2];

  // Calculate magnitudes
  float magBefore = sqrt(sq(before[0]) + sq(before[1]) + sq(before[2]));
  float magAfter = sqrt(sq(after[0]) + sq(after[1]) + sq(after[2]));

  // Calculate angle in radians then convert to degrees
  float angle = acos(dot / (magBefore * magAfter)) * 57.2958;
  return angle;
}

void triggerVibration(uint8_t count, uint16_t duration) {
  for (uint8_t i = 0; i < count; i++) {
    digitalWrite(VIBRATION_PIN, HIGH);
    delay(duration);
    digitalWrite(VIBRATION_PIN, LOW);
    if (i < count-1) delay(200);
  }
}
```