Wearable AI Pin Embedded System Engineer Assignment solution

Hiring Challenge Response

Candidate: Palli Anil Kumar

Date: January 29, 2026

Position: Embedded System Engineer

Table of Contents

Introduction

- This document presents a comprehensive solution to the Wearable AI Pin engineering challenge. The challenge focuses on designing a battery-efficient, privacy-conscious wearable device that enables voice interaction with cloud-based Large Language Models (LLMs).
- The key constraint is achieving 12+ hours of battery life while maintaining always-on listening capability for wake-word detection. This requires careful consideration of power management, memory optimization, and hardware-level privacy features.
- This solution addresses all three hardware questions and provides production-ready firmware code implementing the required state machine logic

Question 1: The Always-Listening Problem

Challenge: The microphone must always be on to catch the wake word ("Hey Computer"), which drains the battery significantly. The proposed design uses a dual-core SoC with:

- **Core A**: High performance, Wi-Fi capable, power hungry
- **Core B**: Low power, slow speed, limited RAM

Solution: Power-Optimized Task Distribution

Core B (Low Power Core) - Always Running:

1. **Wake Word Detection:** Runs a lightweight DSP-based wake word detector continuously. Uses minimal power (1-5mW) with optimized algorithms like keyword spotting neural networks.

2. **Audio Buffering:** Maintains a 10-second circular buffer in local SRAM to capture audio context before the wake word or button press.

3. **Button Monitoring:** Polls the touch sensor via low-power I2C interface.

4. **System Orchestration:** When wake word detected or button pressed, sends interrupt to wake Core A from deep sleep.

5. **Power Monitoring:** Checks battery level and can shut down Core A if battery critical.

Core A (High Performance Core) - Activated Only When Needed:

1. **Cloud Upload:** Handles WiFi connection and streaming audio data to the cloud LLM backend.

2. **Audio Encoding:** Compresses audio using Opus or AAC codec before transmission to reduce bandwidth and power consumption.

3. **Response Handling:** Receives and plays back Text-to-Speech (TTS) audio from the LLM.

4. **Deep Sleep:** Enters ultra-low-power sleep mode when idle, consuming <1mW.

Power Consumption Analysis:

- **Idle State (Core B only):** ~5mW → 12+ hours battery life achievable
- **Active State (both cores):** ~505mW → Limited duration, triggered only by user interaction
- **Average Power:** Assuming 5 minutes of active use per day: ~8mW average → 15+ hours battery life

This architecture maximizes battery life by keeping the power-hungry Core A in deep sleep 99% of the time, while Core B handles all continuous monitoring tasks with minimal power draw

Question 2: Memory Calculation (Audio Buffer)

Requirements:

- Sample Rate: 16 kHz
- Bit Depth: 16-bit (2 bytes per sample)
- Channels: Mono
- Duration: 10 seconds
- Available SRAM: 256 KB

Calculation:

Total buffer size = Sample Rate × Bytes per Sample × Duration

Total buffer size = 16,000 Hz × 2 bytes × 10 seconds

Total buffer size = 320,000 bytes

Total buffer size = 320 KB

Parameter	Value
Required Buffer Size	320KB
Available SRAM	256KB
Shortfall	64KB
Result	DOES NOT FIT

Recommended Solutions:

1. **Reduce Buffer Duration:** Use an 8-second buffer instead of 10 seconds. This requires exactly 256 KB (16,000 × 2 × 8 = 256,000 bytes), fitting perfectly in available SRAM.

2. **External Memory:** Add external SPI flash or PSRAM (pseudo-static RAM) for the audio buffer. This adds minimal cost (~$0.50) and allows flexible buffer sizing.

3. **Streaming Architecture (Recommended):** Use a smaller 2-3 second buffer (64 KB) and stream audio directly to the cloud in real-time. This is more memory-efficient and privacy-friendly as audio isn't stored locally.

4. **Audio Compression:** Use ADPCM (Adaptive Differential Pulse Code Modulation) encoding at 4 bits per sample instead of 16 bits. This reduces the buffer size to 160 KB (16,000 × 0.5 × 10 = 80 KB for 4-bit encoding).

Best Practice: Implement solution #3 (streaming) with a 2-3 second buffer for context. This optimizes both memory usage and privacy.

Question 3: Privacy Hardware Switch

Requirement: Implement a "Hard Mute" switch that physically cuts microphone power to prevent secret recording. The switch must work even if software crashes.

Hardware Design:

The hardware mute switch must be placed in series with the microphone's power supply, creating a physical break in the circuit that cannot be overridden by software.

Circuit Schematic:

VDD (3.3V)

|

|----[Physical Switch]----+

|

[MIC]

|

GND

Critical Design Points:

1. **No GPIO Control:** The switch has NO connection to any GPIO pin or software-controllable component.
2. **Series Configuration:** When open, the switch creates a complete power disconnect. The microphone receives 0V and cannot function.
3. **Crash-Proof:** Even if the MCU crashes, hangs, or is compromised, the physical switch state is preserved. Software cannot turn the mic back on.
4. **Visual Indicator:** An LED should be hardwired in parallel with the microphone power (with appropriate current limiting) to show the true hardware state

Aspect	Correct(Hardware)	Wrong(Software)
Switch Location	In power rail	Connected to GPIO
Software Control	None-physically impossible	Can be overridden
Crash Resistance	Always works	Fails if software crashes
Privacy Guarantee	Absolute	Vulnerable to bugs/hucks
Implementation	Physical circuit break	Software flag/register

Additional Privacy Measures:

- Hardware LED indicator showing true microphone power state (not software-controlled)
- End-to-end encryption of audio data before WiFi transmission
- No local storage of unencrypted audio
- Transparent privacy policy and open-source firmware option

Part 2: Firmware Logic (C Code)

The following C code implements a robust state machine for the AI Pin device. The code follows all specified constraints: switch-statement based, no floating-point operations, 30-second safety timeout, and proper battery level monitoring

```
#include <stdint.h>
#include <stdbool.h>
 // State definitions
typedef enum {
STATE_IDLE,
STATE_LISTENING,
STATE_ERROR }
DeviceState;
// LED color definitions
#define LED_OFF 0
#define LED_GREEN 1
#define LED_RED 2
```

```
// Global state variables static
DeviceState current_state = STATE_IDLE;
static uint32_t button_hold_start_time = 0;
static const uint32_t MAX_HOLD_TIME_MS = 30000; // 30 seconds
// Mock function prototypes (provided by hardware abstraction layer)
bool is_button_pressed(void);
int get_battery_level(void);
void set_led_color(int color);
void start_audio_stream(void);
void stop_audio_stream(void);
uint32_t get_system_time_ms(void); // Returns time in milliseconds
void device_state_machine(void) { int battery = get_battery_level();
bool button_pressed = is_button_pressed();
uint32_t current_time = get_system_time_ms();
switch(current_state) { case STATE_IDLE:
// IDLE State: LED OFF, Audio OFF, minimal power consumption
set_led_color(LED_OFF);
stop_audio_stream();
button_hold_start_time = 0;
// Transition: Button pressed -> Check battery before proceeding
if (button_pressed) {
if (battery > 10) {
// Battery sufficient - enter LISTENING state
current_state = STATE_LISTENING;
button_hold_start_time = current_time; }
else {
// Battery too low - enter ERROR state
current_state = STATE_ERROR; } }
break;
```

```
case STATE_LISTENING:
// LISTENING State: LED GREEN, Audio streaming active
set_led_color(LED_GREEN);
start_audio_stream();
// Safety check: Battery level must stay above 10%
if (battery < 10) {
current_state = STATE_ERROR;
break; }
// Safety check: Prevent overheating from stuck button
if (button_pressed) {
uint32_t hold_duration = current_time - button_hold_start_time;
if (hold_duration > MAX_HOLD_TIME_MS) {
// Force reset to IDLE after 30 seconds
stop_audio_stream();
set_led_color(LED_OFF);
current_state = STATE_IDLE;
button_hold_start_time = 0;
break; } }
// Hold-to-talk: Return to IDLE when button released
if (!button_pressed) {
stop_audio_stream();
set_led_color(LED_OFF);
current_state = STATE_IDLE;
button_hold_start_time = 0; }
break;
case STATE_ERROR:
// ERROR State: LED RED, Audio OFF, waiting for recovery
set_led_color(LED_RED);
stop_audio_stream();
```

```
// Recovery: Return to IDLE when battery recharged AND button released
if (battery > 15 && !button_pressed) {
current_state = STATE_IDLE; }
break;
default:
// Invalid state - reset to IDLE for safety
current_state = STATE_IDLE; break; } }
int main(void) {
while(1) {
device_state_machine(); }
 return 0; }
```

Code Implementation Features

Requirements Compliance:

- Switch-statement based state machine (as specified)
- No floating-point arithmetic (integer-only operations)
- 30-second safety timeout to prevent overheating
- Battery monitoring with 10% threshold for operation
- Hold-to-talk interface requiring continuous button press

Software Engineering Best Practices:

- Clear, readable code with comprehensive comments
- Proper state transition handling with cleanup
- Defensive programming with default case
- Well-defined function interfaces (HAL abstraction)
- Enumerated types for type safety
- Constants defined with #define for easy tuning

Safety & Reliability Features:

- Multiple safety checks prevent unsafe operation
- Graceful error handling and recovery
- No memory leaks or resource leaks
- Deterministic behavior in all scenarios
- Protected against button hardware failures