PARKING SENSOR CHALLENGE REPORT

Passoni Gabriele

ABSTRACT

The challenge I encountered revolved around developing a parking sensor capable of determining the availability of parking spaces. This was achieved by utilizing an ESP32-v4 controller and an HC-SR04 ultrasonic distance sensor, both simulated on the online platform Wokwi. I devised two slightly different versions of the project to address specific aspects:

1) The first version was dedicated to debugging and verifying the proper functionality of our IoT device.
   [DEBUG version link]
2) The second version was tailored for timing profiling its performance.
   [TIME version link]

CODE EXPLANATION

Libraries

To tackle the challenge, I needed to incorporate two libraries: the first one, `Wifi.h`, is essential for managing the WiFi functionalities of the controller, which, in this instance, has been configured as an access point; the second one, `esp_now.h`, is vital for utilizing the ESP NOW communication protocol.

Connecting the sensor and using it

The HC-SR04 sensor requires correct connection to the controller using 4 pins: the first pin is for a 5V power supply, the second is the ground pin, the third is the trigger pin (sensor input), and the fourth and last pin is the echo pin (sensor output). These latter two pins have been respectively connected to pins #33 and #32, which are GPIO dedicated to input and output. The sensor operates according to the following rules:

a) Measurement is initiated by a high impulse of at least 10 microseconds.
b) The sensor produces a high impulse of duration x, from which the distance in centimeters can be obtained using the formula: $distance = x/58$.

The sensor is utilized through the function `sense()`.

ESP NOW and WiFi

To establish communication with the sink node, I employed the ESP NOW protocol, which facilitates communication with one or multiple devices (up to 20) supporting the same technology, utilizing MAC addresses and the WiFi channel. I managed the WiFi channel using the primitive `WiFi.mode(wifi_mode)`, which can activate it (as an access point) or deactivate it. To leverage ESP NOW, I followed these steps:

a) Initialization of ESP NOW with `esp_now_init()`.
b) Initialization of an `esp_now_peer_info_t` struct using default parameters (no encryption and standard WiFi channel 0) and the broadcast MAC address.
c) Adding the sink node to the contact list with `esp_now_add_peer(&peerInfo)`.

Following this setup, I were able to transmit messages to the sink using the primitive `esp_now_send(addr, msg, size)`. It's noteworthy that in the first version, I also associated a callback function upon message reception. This allowed us to assess the correct functioning of the IoT device, as each message sent was also received using the broadcast MAC address and printed in the log (see `onDataRecv(mac, data, len)`).

Deep sleep triggering and timed wake-up

To minimize energy consumption, I implemented a duty cycle using the built-in functions of the ESP32. Following the formula specified in the challenge, I enforced a deep sleep period of 9 seconds and detected a duty cycle of about 200 ms. To achieve this, I configured a timer to wake up the device using the primitive `esp_sleep_enable_timer_wakeup(sleeping_time_us)` and then initiated the deep sleep mode using `esp_deep_sleep_start()`.

Final considerations on code

a) In the second version, aimed at profiling our device's timing, I employed the `micros()` primitive to measure the time differences at various points in the execution.
b) In both versions, the `loop()` function is bypassed since the device calls the `setup()` function upon waking up from deep sleep. Therefore, I effectively looped within the `setup()` function instead of the standard `loop()`.

POSSIBLE IMPROVEMENTS

a) Dynamic WiFi band adjustment:
   Currently, the project utilizes the default WiFi band. An enhancement would involve implementing the capability to dynamically adjust the WiFi band based on environmental conditions. For instance, during periods of network congestion, the device could automatically switch to a less utilized band to ensure a more stable and reliable connection.
b) Notify only changes of state:
   Presently, the device sends the "occupied" or "free" signal every time it is activated, irrespective of whether the parking state has changed. An optimization would entail the device measuring the distance from the object and transmitting the signal only if the parking state changes from the previously detected state. This approach minimizes unnecessary WiFi activation and transmission of redundant messages, consequently extending battery life.
c) Dynamic and scheduled deep sleep state duration:
   To enhance energy efficiency, I propose dynamic adjustment of deep sleep duration based on recent parking state change history. For instance, if the device observes sustained occupancy over several measurement cycles, it would prolong deep sleep to conserve energy. Conversely, frequent state changes would prompt shorter deep sleep periods, ensuring responsiveness to environmental shifts. Additionally, scheduled deep sleep states can be enforced based on seasonality and parking lot hours. For instance, during closed periods, devices would remain in deep sleep until opening, further optimizing energy usage.
d) Tailored transmission power:
   Currently, our device operates with a default transmission power of 19.5 dBm. However, I propose adjusting this setting based on the distance from the sink node and the measured environmental noise. By customizing transmission power, I aim to improve both Bit Error Rate (BER) and Packet Error Rate (PER), while also conserving energy. This optimization is particularly beneficial in scenarios where requirements are less stringent, such as when the sink node is nearby or environmental noise is minimal. Adjusting transmission power dynamically allows us to optimize performance and energy usage according to specific conditions.