```
/**
 * Smart Parking System - Arduino IoT Cloud Implementation
 *
 * Features:
 * - Dual parking slot monitoring with IR sensors
 * - Automatic gate control with ultrasonic proximity detection
 * - Visual status indicators (LEDs)
 * - Time-limited parking enforcement
 * - Cloud connectivity with Arduino IoT Cloud
 * - Optimized serial output with essential information only
 * - Smooth servo gate operation
 */

#include <ESP8266WiFi.h>
#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>
#include <Servo.h>

// Network credentials and IoT Cloud connectivity
#define WIFI_SSID "Chirag"
#define WIFI_PASS "forgetit"

// Your Arduino IoT Cloud credentials
const char DEVICE_ID[] = "3c723647-8f91-40c6-ac74-f6ea08f20064";
const char THING_ID[] = "04eead99-5d79-4c3e-950f-7e07f0e8feb4";
const char SECRET_KEY[] = "yx25Xo6trAoqlf6iafPLahpue";

// Pin definitions
#define PIN_SLOT1_IR D2
#define PIN_SLOT2_IR D3
#define PIN_ULTRASONIC_TRIG D5
#define PIN_ULTRASONIC_ECHO D6
#define PIN_SERVO D1
#define PIN_LED_GREEN D7
#define PIN_LED_RED D8

// System constants
#define PARKING_TIME_LIMIT_MS 60000      // 60 seconds (1 minute)
#define GATE_OPEN_ANGLE 90               // 90-degree angle for open
barrier
#define GATE_CLOSED_ANGLE 0              // Servo angle for closed
barrier
#define SERVO_STEP_DELAY 50              // Increased for smoother
movement
#define SERVO_STEP_SIZE 2                // Move 2 degrees at a time
for smoother motion
#define GATE_OPEN_DURATION_MS 2000       // Time gate stays open (2
seconds)
#define CAR_DETECTION_THRESHOLD_CM 15    // Distance threshold for car
detection
```

```
#define VALID_DISTANCE_MIN_CM 2          // Minimum valid distance
reading
#define VALID_DISTANCE_MAX_CM 400        // Maximum valid distance
reading
#define DEFAULT_DISTANCE_CM 100          // Default distance when
sensor reading fails
#define STATUS_UPDATE_INTERVAL_MS 10000  // Status update interval (10
seconds)
#define SENSOR_CHECK_INTERVAL_MS 200     // Check sensors every 200ms

// System operation variables
Servo gateServo;
bool slot1Occupied = false;
bool slot2Occupied = false;
bool parkingFull = false;
int carCount = 0;
float distanceCm = DEFAULT_DISTANCE_CM;
bool carDetected = false;

// Time tracking variables
unsigned long slot1OccupiedSince = 0;
unsigned long slot2OccupiedSince = 0;
unsigned long lastGateOperation = 0;
unsigned long lastStatusUpdate = 0;
unsigned long lastSensorCheck = 0;
bool slot1AlertSent = false;
bool slot2AlertSent = false;
bool carDetectedPrevious = false;
bool gateOperational = false;
int currentServoPos = 0;

// WiFi status variables
String wifiStatus = "Disconnected";
String ipAddress = "0.0.0.0";

// Arduino IoT Cloud variables (limited to 5)
bool parkingStatus;                      // true = full, false =
available
int occupiedSlots;                       // 0, 1, or 2
bool entranceOccupied;                   // true = car at entrance,
false = clear
int timeRemainingSlot1;                  // Time remaining or -1 if
empty
int timeRemainingSlot2;                  // Time remaining or -1 if
empty

// Arduino IoT Cloud connection handler
WiFiConnectionHandler ArduinoIoTConnectionHandler(WIFI_SSID,
WIFI_PASS);
```

```
/**
 * Initialize the IoT Cloud variables and their properties
 */
void initProperties() {
  ArduinoCloud.setThingId(THING_ID);
  ArduinoCloud.setDeviceId(DEVICE_ID);
  ArduinoCloud.setSecretDeviceKey(SECRET_KEY);

  ArduinoCloud.addProperty(parkingStatus, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(occupiedSlots, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(entranceOccupied, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(timeRemainingSlot1, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(timeRemainingSlot2, READ, ON_CHANGE, NULL);
}

/**
 * Initialize and test servo motor
 */
void initServo() {
  gateServo.attach(PIN_SERVO, 500, 2400);
  delay(500);
  currentServoPos = GATE_CLOSED_ANGLE;
  moveServoSmoothly(GATE_CLOSED_ANGLE);
  gateOperational = true;
}

/**
 * Move servo smoothly in steps
 * @param targetAngle The target angle to move to
 */
void moveServoSmoothly(int targetAngle) {
  // Validate angle is within 0-90 range
  targetAngle = constrain(targetAngle, 0, 90);

  // Determine direction
  int step = (targetAngle > currentServoPos) ? SERVO_STEP_SIZE : -
SERVO_STEP_SIZE;

  // Move in small steps
  while (abs(currentServoPos - targetAngle) > abs(step)) {
    currentServoPos += step;
    gateServo.write(currentServoPos);
    delay(SERVO_STEP_DELAY);  // Increased delay for smoother movement
  }

  // Final step to exact position
  gateServo.write(targetAngle);
  currentServoPos = targetAngle;
}
```

```
/**
 * Update WiFi connection status variables
 */
void updateWiFiStatus() {
  wl_status_t status = WiFi.status();

  switch (status) {
    case WL_CONNECTED:
      wifiStatus = "Connected";
      ipAddress = WiFi.localIP().toString();
      break;
    case WL_IDLE_STATUS:
      wifiStatus = "Idle";
      ipAddress = "0.0.0.0";
      break;
    case WL_DISCONNECTED:
      wifiStatus = "Disconnected";
      ipAddress = "0.0.0.0";
      break;
    case WL_CONNECT_FAILED:
      wifiStatus = "Connection Failed";
      ipAddress = "0.0.0.0";
      break;
    case WL_NO_SSID_AVAIL:
      wifiStatus = "SSID Not Available";
      ipAddress = "0.0.0.0";
      break;
    default:
      wifiStatus = "Unknown (" + String(status) + ")";
      ipAddress = "0.0.0.0";
  }
}

/**
 * Measure distance using ultrasonic sensor
 * @return Distance in centimeters
 */
float measureDistance() {
  digitalWrite(PIN_ULTRASONIC_TRIG, LOW);
  delayMicroseconds(2);

  digitalWrite(PIN_ULTRASONIC_TRIG, HIGH);
  delayMicroseconds(10);
  digitalWrite(PIN_ULTRASONIC_TRIG, LOW);

  unsigned long duration = pulseIn(PIN_ULTRASONIC_ECHO, HIGH, 30000);

  if (duration == 0) {
    return DEFAULT_DISTANCE_CM;
  }
  float distance = duration * 0.034 / 2.0;

  if (distance < VALID_DISTANCE_MIN_CM || distance >
VALID_DISTANCE_MAX_CM) {
    return DEFAULT_DISTANCE_CM;
  }

  return distance;
}

/**
 * Update LED indicators based on parking status
 */
void updateLedIndicators() {
  bool anySlotAvailable = !slot1Occupied || !slot2Occupied;

  digitalWrite(PIN_LED_GREEN, anySlotAvailable ? HIGH : LOW);
  digitalWrite(PIN_LED_RED, parkingFull ? HIGH : LOW);
}

/**
 * Control gate operation
 * @param open True to open gate, false to close
 */
void controlGate(bool open) {
  if (!gateOperational) {
    Serial.println("WARNING: Gate not operational");
    return;
  }

  if (open) {
    Serial.println("* Gate: Opening");
    moveServoSmoothly(GATE_OPEN_ANGLE);
    lastGateOperation = millis();
  } else {
    Serial.println("* Gate: Closing");
    moveServoSmoothly(GATE_CLOSED_ANGLE);
  }
}

/**
 * Update parking times and send alerts if necessary
 */
void checkParkingTimeLimits() {
  unsigned long currentTime = millis();

  // Check slot 1
  if (slot1Occupied && !slot1AlertSent &&
      (currentTime - slot1OccupiedSince >= PARKING_TIME_LIMIT_MS)) {
```

```
    Serial.println("! ALERT: Slot 1 time limit exceeded");
    slot1AlertSent = true;
  } else if (!slot1Occupied) {
    slot1AlertSent = false;
  }

  // Check slot 2
  if (slot2Occupied && !slot2AlertSent &&
      (currentTime - slot2OccupiedSince >= PARKING_TIME_LIMIT_MS)) {
    Serial.println("! ALERT: Slot 2 time limit exceeded");
    slot2AlertSent = true;
  } else if (!slot2Occupied) {
    slot2AlertSent = false;
  }

  // Update time remaining for cloud variables
  if (slot1Occupied) {
    unsigned long timeOccupied = (currentTime - slot1OccupiedSince) /
1000;
    long timeLeft = (PARKING_TIME_LIMIT_MS / 1000) - timeOccupied;
    timeRemainingSlot1 = timeLeft > 0 ? timeLeft : 0;
  } else {
    timeRemainingSlot1 = -1;
  }

  if (slot2Occupied) {
    unsigned long timeOccupied = (currentTime - slot2OccupiedSince) /
1000;
    long timeLeft = (PARKING_TIME_LIMIT_MS / 1000) - timeOccupied;
    timeRemainingSlot2 = timeLeft > 0 ? timeLeft : 0;
  } else {
    timeRemainingSlot2 = -1;
  }
}

/**
 * Update cloud variables
 */
void updateCloudVariables() {
  parkingStatus = parkingFull;
  occupiedSlots = carCount;
  entranceOccupied = carDetected;
  // Time remaining is updated in checkParkingTimeLimits()
}

/**
 * Print concise system status with just essential information
 */
void printSystemStatus() {
  // Update WiFi status
  updateWiFiStatus();

  Serial.println("\n------- SMART PARKING STATUS -------");
  Serial.print("WiFi: ");
  Serial.print(wifiStatus);
  if (wifiStatus == "Connected") {
    Serial.print(" (");
    Serial.print(ipAddress);
    Serial.print(", ");
    Serial.print(WiFi.RSSI());
    Serial.println(" dBm)");
  } else {
    Serial.println();
  }

  Serial.print("Cloud: ");
  Serial.println(ArduinoCloud.connected() ? "Connected" :
 "Disconnected");

  Serial.println("\nParking:");
  Serial.print("Slot 1: ");
  Serial.print(slot1Occupied ? "Occupied" : "Empty");
  if (slot1Occupied) {
    int timeLeft = timeRemainingSlot1;
    Serial.print(" (");
    if (timeLeft > 0) {
      Serial.print(timeLeft);
      Serial.print("s left)");
    } else {
      Serial.print("Time exceeded)");
    }
  }
  Serial.println();

  Serial.print("Slot 2: ");
  Serial.print(slot2Occupied ? "Occupied" : "Empty");
  if (slot2Occupied) {
    int timeLeft = timeRemainingSlot2;
    Serial.print(" (");
    if (timeLeft > 0) {
      Serial.print(timeLeft);
      Serial.print("s left)");
    } else {
      Serial.print("Time exceeded)");
    }
  }
  Serial.println();

  Serial.print("Car at entrance: ");
  Serial.println(carDetected ? "Yes" : "No");
  Serial.print("Status: ");
  if (parkingFull) {
    Serial.println("FULL (Red LED on)");
  } else {
    Serial.println("SPACES AVAILABLE (Green LED on)");
  }

  Serial.print("Gate: ");
  Serial.println(currentServoPos > 0 ? "OPEN" : "CLOSED");
  Serial.println("------------------------------------");
}

/**
 * Print brief event notification
 */
void printEvent(const char* eventType) {
  Serial.print("* ");
  Serial.print(eventType);
  Serial.println();
}

/**
 * Initialize hardware components
 */
void initHardware() {
  pinMode(PIN_SLOT1_IR, INPUT);
  pinMode(PIN_SLOT2_IR, INPUT);
  pinMode(PIN_ULTRASONIC_TRIG, OUTPUT);
  pinMode(PIN_ULTRASONIC_ECHO, INPUT);
  pinMode(PIN_LED_GREEN, OUTPUT);
  pinMode(PIN_LED_RED, OUTPUT);

  digitalWrite(PIN_LED_GREEN, HIGH);
  digitalWrite(PIN_LED_RED, LOW);

  initServo();
}

/**
 * System setup
 */
void setup() {
  Serial.begin(9600);
  delay(500);  // Ensure serial is ready

  Serial.println("\n===================================");
  Serial.println("  SMART PARKING SYSTEM - IoT CLOUD  ");
  Serial.println("===================================");
  Serial.println("Initializing...");
  initProperties();
  ArduinoCloud.begin(ArduinoIoTConnectionHandler);

  parkingStatus = false;
  occupiedSlots = 0;
  entranceOccupied = false;
  timeRemainingSlot1 = -1;
  timeRemainingSlot2 = -1;

  initHardware();
  Serial.println("Hardware initialized");

  // Flash LEDs to indicate successful startup
  for (int i = 0; i < 3; i++) {
    digitalWrite(PIN_LED_GREEN, HIGH);
    digitalWrite(PIN_LED_RED, HIGH);
    delay(200);
    digitalWrite(PIN_LED_GREEN, LOW);
    digitalWrite(PIN_LED_RED, LOW);
    delay(200);
  }

  digitalWrite(PIN_LED_GREEN, HIGH);
  digitalWrite(PIN_LED_RED, LOW);

  Serial.println("System ready!");
  lastStatusUpdate = millis();
  lastSensorCheck = millis();
}

/**
 * Main system loop
 */
void loop() {
  ArduinoCloud.update();

  unsigned long currentTime = millis();

  // Check sensors on a fixed interval instead of every loop
  if (currentTime - lastSensorCheck >= SENSOR_CHECK_INTERVAL_MS) {
    // Measure distance and detect cars
    distanceCm = measureDistance();
    bool newCarDetected = (distanceCm < CAR_DETECTION_THRESHOLD_CM &&
distanceCm > 0);

    // Car newly detected at entrance
    if (newCarDetected && !carDetectedPrevious) {
      if (!parkingFull && gateOperational) {
        printEvent("Car at entrance - Opening gate");
        controlGate(true);
      } else if (parkingFull) {
        printEvent("Car at entrance - Parking full, gate remains
closed");
      }
    }

    // Car leaving detection
    if (!newCarDetected && carDetectedPrevious) {
      printEvent("Car exiting entrance area");
    }

    // Close gate after delay if it's been open
    if (lastGateOperation > 0 && gateOperational &&
        (currentTime - lastGateOperation >= GATE_OPEN_DURATION_MS)) {
      controlGate(false);
      lastGateOperation = 0;
    }

    // Update detection state
    carDetected = newCarDetected;
    carDetectedPrevious = newCarDetected;

    // Read IR sensors
    bool newSlot1Occupied = (digitalRead(PIN_SLOT1_IR) == LOW);
    bool newSlot2Occupied = (digitalRead(PIN_SLOT2_IR) == LOW);

    // Check for slot 1 status change
    if (!slot1Occupied && newSlot1Occupied) {
      slot1OccupiedSince = currentTime;
      printEvent("Car parked in Slot 1");
    }

    if (slot1Occupied && !newSlot1Occupied) {
      printEvent("Car left Slot 1");
      controlGate(true);
      lastGateOperation = currentTime;
    }

    // Check for slot 2 status change
    if (!slot2Occupied && newSlot2Occupied) {
      slot2OccupiedSince = currentTime;
      printEvent("Car parked in Slot 2");
    }

    if (slot2Occupied && !newSlot2Occupied) {
      printEvent("Car left Slot 2");
      controlGate(true);
      lastGateOperation = currentTime;
    }
    // Update slot status
    slot1Occupied = newSlot1Occupied;
    slot2Occupied = newSlot2Occupied;

    // Update car count
    carCount = (slot1Occupied ? 1 : 0) + (slot2Occupied ? 1 : 0);

    // Update parking full status
    bool newParkingFull = slot1Occupied && slot2Occupied;
    if (newParkingFull && !parkingFull) {
      printEvent("Parking is now FULL");
    } else if (!newParkingFull && parkingFull) {
      printEvent("Parking spaces now AVAILABLE");
    }
    parkingFull = newParkingFull;

    // Update indicators
    updateLedIndicators();

    // Check time limits
    checkParkingTimeLimits();

    // Update cloud variables
    updateCloudVariables();

    lastSensorCheck = currentTime;
  }

  // Print status at regular intervals
  if (currentTime - lastStatusUpdate >= STATUS_UPDATE_INTERVAL_MS) {
    printSystemStatus();
    lastStatusUpdate = currentTime;
  }

  // Brief delay for stability (shorter since we're already pacing
with intervals)
  delay(10);
}
```