#include <SoftwareSerial.h>

#include <EEPROM.h>

// Pin definitions

#define LEFT\_MOTOR\_PIN1 D1

#define LEFT\_MOTOR\_PIN2 D2

#define RIGHT\_MOTOR\_PIN1 D3

#define RIGHT\_MOTOR\_PIN2 D4

#define TRIG\_PIN D5

#define ECHO\_PIN D6

#define MINE\_SENSOR\_PIN A0

#define BT\_TX D7

#define BT\_RX D8

int sensorValue;

// Grid configuration

const int GRID\_WIDTH = 10;

const int GRID\_HEIGHT = 10;

const int UNIT\_LENGTH = 10; // 10 cm per unit

const int MOVE\_TIME = 1000; // Time to move one unit

// Sensor thresholds

const int MINE\_THRESHOLD = 250;

const int OBSTACLE\_DISTANCE = 10; // cm

// Grid states

#define EMPTY 0

#define MINE 1

#define OBSTACLE 2

#define VISITED 3

// Grid mapping

int grid[GRID\_HEIGHT][GRID\_WIDTH];

int currentX = 0;

int currentY = 0;

int direction = 0; // 0=North, 1=East, 2=South, 3=West

bool explorationComplete = false;

bool completionMessageSent = false;

bool missionComplete = false; // New flag to track mission completion

// Movement vectors

int dx[] = {0, 1, 0, -1}; // North, East, South, West

int dy[] = {-1, 0, 1, 0};

SoftwareSerial bluetooth(BT\_RX, BT\_TX);

void setup() {

Serial.begin(38400);

bluetooth.begin(38400);

// Initialize pins

pinMode(LEFT\_MOTOR\_PIN1, OUTPUT);

pinMode(LEFT\_MOTOR\_PIN2, OUTPUT);

pinMode(RIGHT\_MOTOR\_PIN1, OUTPUT);

pinMode(RIGHT\_MOTOR\_PIN2, OUTPUT);

pinMode(TRIG\_PIN, OUTPUT);

pinMode(ECHO\_PIN, INPUT);

pinMode(MINE\_SENSOR\_PIN, INPUT);

// Initialize grid

initializeGrid();

delay(2000);

Serial.println("Advanced Mine Detection Rover Starting...");

sendMapUpdate();

}

void loop() {

// Check current position for mine first

if (checkMine()) {

handleMineDetection();

delay(1000);

return;

}

// Mark current position as visited

if (grid[currentY][currentX] == EMPTY) {

grid[currentY][currentX] = VISITED;

}

// Main exploration logic

if (!explorationComplete) {

exploreGrid();

// Check if exploration is now complete

if (explorationComplete && !completionMessageSent) {

sendCompletionMessage();

completionMessageSent = true;

}

} else {

// Exploration complete, return to start

returnToStart();

}

sendMapUpdate();

delay(500);

}

void initializeGrid() {

for (int i = 0; i < GRID\_HEIGHT; i++) {

for (int j = 0; j < GRID\_WIDTH; j++) {

grid[i][j] = EMPTY;

}

}

grid[currentY][currentX] = VISITED;

}

bool checkMine() {

sensorValue = analogRead(MINE\_SENSOR\_PIN);

return sensorValue < MINE\_THRESHOLD;

}

bool checkObstacle() {

digitalWrite(TRIG\_PIN, LOW);

delayMicroseconds(2);

digitalWrite(TRIG\_PIN, HIGH);

delayMicroseconds(10);

digitalWrite(TRIG\_PIN, LOW);

long duration = pulseIn(ECHO\_PIN, HIGH);

float distance = (duration \* 0.034) / 2;

return (distance < OBSTACLE\_DISTANCE && distance > 0);

}

void handleMineDetection() {

Serial.println("MINE DETECTED! Avoiding and mapping...");

// Mark mine location

grid[currentY][currentX] = MINE;

// Stop and plan avoidance

stop();

delay(500);

// Smart mine avoidance with inward movement priority

if (!avoidMineInward()) {

// If inward movement fails, use spiral avoidance

avoidMineSpiral();

}

}

bool avoidMineInward() {

int centerX = GRID\_WIDTH / 2;

int centerY = GRID\_HEIGHT / 2;

// Calculate direction toward center

int bestDirection = getBestDirectionToCenter();

// Try to move in the best direction

for (int attempt = 0; attempt < 4; attempt++) {

int tryDir = (bestDirection + attempt) % 4;

// Turn to face this direction

turnToDirection(tryDir);

// Check if path is clear

if (!checkObstacle() && canMoveInDirection(tryDir)) {

moveForward();

updatePosition();

return true;

}

}

return false;

}

void avoidMineSpiral() {

Serial.println("Using spiral avoidance pattern");

// Try directions in spiral order: right, back, left, forward

int spiralOrder[] = {

(direction + 1) % 4, // Right

(direction + 2) % 4, // Back

(direction + 3) % 4, // Left

direction // Forward

};

for (int i = 0; i < 4; i++) {

turnToDirection(spiralOrder[i]);

if (!checkObstacle() && canMoveInDirection(spiralOrder[i])) {

moveForward();

updatePosition();

// After moving, try to orient toward unexplored area

orientTowardUnexplored();

return;

}

}

// If completely stuck, mark as obstacle and backtrack

Serial.println("Completely blocked - backtracking");

backtrack();

}

int getBestDirectionToCenter() {

int centerX = GRID\_WIDTH / 2;

int centerY = GRID\_HEIGHT / 2;

int bestDir = direction;

float minDistance = calculateDistance(currentX, currentY, centerX, centerY);

for (int dir = 0; dir < 4; dir++) {

int nextX = currentX + dx[dir];

int nextY = currentY + dy[dir];

if (isValidPosition(nextX, nextY)) {

float distance = calculateDistance(nextX, nextY, centerX, centerY);

if (distance < minDistance) {

minDistance = distance;

bestDir = dir;

}

}

}

return bestDir;

}

void exploreGrid() {

// Check for obstacles ahead

if (checkObstacle()) {

handleObstacleDetection();

return;

}

// Priority-based exploration

int nextDirection = chooseExplorationDirection();

if (nextDirection != -1) {

turnToDirection(nextDirection);

if (canMoveInDirection(nextDirection)) {

moveForward();

updatePosition();

} else {

// Path blocked, try alternative

findAlternativePath();

}

} else {

// No unexplored areas accessible, check if truly complete

if (isGridExplorationComplete()) {

explorationComplete = true;

Serial.println("Grid exploration complete!");

} else {

// Try to find a path to any remaining unexplored areas

findAlternativePath();

}

}

}

bool isGridExplorationComplete() {

// Check if all accessible cells have been explored

for (int y = 0; y < GRID\_HEIGHT; y++) {

for (int x = 0; x < GRID\_WIDTH; x++) {

if (grid[y][x] == EMPTY) {

// Found an unexplored cell, check if it's reachable

if (isReachable(x, y)) {

return false; // Still have reachable unexplored areas

}

}

}

}

return true; // All reachable areas explored

}

bool isReachable(int targetX, int targetY) {

// Simple reachability check - can we find a path of non-obstacle cells?

// This is a simplified version; you could implement proper pathfinding if needed

// Check if there's at least one adjacent cell that's not an obstacle

for (int dir = 0; dir < 4; dir++) {

int adjX = targetX + dx[dir];

int adjY = targetY + dy[dir];

if (isValidPosition(adjX, adjY) && grid[adjY][adjX] != OBSTACLE) {

return true;

}

}

return false;

}

int chooseExplorationDirection() {

// Priority: 1) Unexplored areas, 2) Toward center, 3) Systematic sweep

// Check for adjacent unexplored cells

for (int dir = 0; dir < 4; dir++) {

int nextX = currentX + dx[dir];

int nextY = currentY + dy[dir];

if (isValidPosition(nextX, nextY) && grid[nextY][nextX] == EMPTY) {

return dir;

}

}

// If no adjacent unexplored cells, find path to nearest unexplored area

return findPathToUnexplored();

}

int findPathToUnexplored() {

// Simple pathfinding to nearest unexplored cell

int minDistance = GRID\_WIDTH + GRID\_HEIGHT;

int bestDirection = -1;

for (int y = 0; y < GRID\_HEIGHT; y++) {

for (int x = 0; x < GRID\_WIDTH; x++) {

if (grid[y][x] == EMPTY) {

// Found unexplored cell, calculate best direction

int distance = abs(x - currentX) + abs(y - currentY);

if (distance < minDistance) {

minDistance = distance;

// Determine best direction to reach this cell

if (abs(x - currentX) > abs(y - currentY)) {

bestDirection = (x > currentX) ? 1 : 3; // East or West

} else {

bestDirection = (y > currentY) ? 2 : 0; // South or North

}

}

}

}

}

return bestDirection;

}

void handleObstacleDetection() {

// Mark obstacle ahead

int obstacleX = currentX + dx[direction];

int obstacleY = currentY + dy[direction];

if (isValidPosition(obstacleX, obstacleY)) {

grid[obstacleY][obstacleX] = OBSTACLE;

}

// Find alternative path

findAlternativePath();

}

void findAlternativePath() {

// Try directions in order of preference

int preferences[] = {

(direction + 1) % 4, // Right

(direction + 3) % 4, // Left

(direction + 2) % 4 // Back

};

for (int i = 0; i < 3; i++) {

turnToDirection(preferences[i]);

if (!checkObstacle() && canMoveInDirection(preferences[i])) {

moveForward();

updatePosition();

return;

}

}

// If all directions blocked, backtrack

backtrack();

}

void backtrack() {

Serial.println("Obstacle Found! Backtracking to find alternative route");

// Turn around and move to previously visited cell

turnToDirection((direction + 2) % 4);

if (canMoveInDirection(direction)) {

moveForward();

updatePosition();

}

}

void orientTowardUnexplored() {

int bestDir = findPathToUnexplored();

if (bestDir != -1) {

turnToDirection(bestDir);

}

}

void returnToStart() {

// Simple return to origin

if (currentX == 0 && currentY == 0) {

if (!missionComplete) {

Serial.println("Successfully returned to start position!");

Serial.println("=== MISSION COMPLETE! ===");

bluetooth.println("MISSION\_COMPLETE");

missionComplete = true; // Set mission complete flag

}

stop();

return;

}

// Move toward origin

int toOriginDir = -1;

if (currentX > 0) {

toOriginDir = 3; // West

} else if (currentX < 0) {

toOriginDir = 1; // East

} else if (currentY > 0) {

toOriginDir = 0; // North

} else if (currentY < 0) {

toOriginDir = 2; // South

}

if (toOriginDir != -1) {

turnToDirection(toOriginDir);

if (!checkObstacle() && canMoveInDirection(toOriginDir)) {

moveForward();

updatePosition();

}

}

}

void sendCompletionMessage() {

Serial.println("==================================");

Serial.println(" GRID EXPLORATION COMPLETE! ");

Serial.println("==================================");

Serial.println("All accessible areas have been mapped.");

Serial.println("Mines detected and obstacles identified.");

Serial.println("Now returning to start position...");

Serial.println("==================================");

bluetooth.println("EXPLORATION\_COMPLETE");

bluetooth.println("STATUS:Grid exploration finished");

bluetooth.println("ACTION:Returning to start");

// Print final statistics

printFinalStats();

}

void printFinalStats() {

int mineCount = 0;

int obstacleCount = 0;

int visitedCount = 0;

for (int y = 0; y < GRID\_HEIGHT; y++) {

for (int x = 0; x < GRID\_WIDTH; x++) {

switch (grid[y][x]) {

case MINE: mineCount++; break;

case OBSTACLE: obstacleCount++; break;

case VISITED: visitedCount++; break;

}

}

}

Serial.println("=== FINAL EXPLORATION STATISTICS ===");

Serial.printf("Total cells explored: %d\n", visitedCount);

Serial.printf("Mines detected: %d\n", mineCount);

Serial.printf("Obstacles found: %d\n", obstacleCount);

Serial.printf("Grid coverage: %.1f%%\n", (float)(visitedCount + mineCount) / (GRID\_WIDTH \* GRID\_HEIGHT) \* 100);

Serial.println("=====================================");

}

// Movement functions

void turnToDirection(int targetDir) {

while (direction != targetDir) {

turnRight();

}

}

void turnRight() {

analogWrite(LEFT\_MOTOR\_PIN1, 120);

analogWrite(LEFT\_MOTOR\_PIN2, 0);

analogWrite(RIGHT\_MOTOR\_PIN1, 0);

analogWrite(RIGHT\_MOTOR\_PIN2, 120);

delay(500);

stop();

direction = (direction + 1) % 4;

}

void turnLeft() {

analogWrite(LEFT\_MOTOR\_PIN1, 0);

analogWrite(LEFT\_MOTOR\_PIN2, 120);

analogWrite(RIGHT\_MOTOR\_PIN1, 120);

analogWrite(RIGHT\_MOTOR\_PIN2, 0);

delay(500);

stop();

direction = (direction + 3) % 4;

}

void moveForward() {

analogWrite(LEFT\_MOTOR\_PIN1, 120);

analogWrite(LEFT\_MOTOR\_PIN2, 0);

analogWrite(RIGHT\_MOTOR\_PIN1, 120);

analogWrite(RIGHT\_MOTOR\_PIN2, 0);

delay(MOVE\_TIME);

stop();

}

void stop() {

analogWrite(LEFT\_MOTOR\_PIN1, 0);

analogWrite(LEFT\_MOTOR\_PIN2, 0);

analogWrite(RIGHT\_MOTOR\_PIN1, 0);

analogWrite(RIGHT\_MOTOR\_PIN2, 0);

}

// Utility functions

bool canMoveInDirection(int dir) {

int nextX = currentX + dx[dir];

int nextY = currentY + dy[dir];

return isValidPosition(nextX, nextY) && grid[nextY][nextX] != OBSTACLE;

}

bool isValidPosition(int x, int y) {

return x >= 0 && x < GRID\_WIDTH && y >= 0 && y < GRID\_HEIGHT;

}

void updatePosition() {

currentX += dx[direction];

currentY += dy[direction];

currentX = constrain(currentX, 0, GRID\_WIDTH - 1);

currentY = constrain(currentY, 0, GRID\_HEIGHT - 1);

}

float calculateDistance(int x1, int y1, int x2, int y2) {

return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));

}

void sendMapUpdate() {

// Only send map updates if mission is not complete

if (missionComplete) {

return;

}

bluetooth.println("MAP\_START");

bluetooth.print("GRID\_SIZE:");

bluetooth.print(GRID\_WIDTH);

bluetooth.print(",");

bluetooth.println(GRID\_HEIGHT);

bluetooth.print("ROVER\_POS:");

bluetooth.print(currentX);

bluetooth.print(",");

bluetooth.print(currentY);

bluetooth.print(",");

bluetooth.println(direction);

// Send all grid data

for (int y = 0; y < GRID\_HEIGHT; y++) {

for (int x = 0; x < GRID\_WIDTH; x++) {

if (grid[y][x] != EMPTY) {

bluetooth.print("CELL:");

bluetooth.print(x);

bluetooth.print(",");

bluetooth.print(y);

bluetooth.print(",");

bluetooth.println(grid[y][x]);

}

}

}

bluetooth.println("MAP\_END");

// Debug output

printGridToSerial();

}

void printGridToSerial() {

// Only print grid to serial if mission is not complete

if (missionComplete) {

return;

}

Serial.println("\n=== Current Grid Status ===");

for (int y = 0; y < GRID\_HEIGHT; y++) {

for (int x = 0; x < GRID\_WIDTH; x++) {

if (x == currentX && y == currentY) {

Serial.print("R ");

} else {

switch (grid[y][x]) {

case EMPTY: Serial.print(". "); break;

case MINE: Serial.print("M "); break;

case OBSTACLE: Serial.print("O "); break;

case VISITED: Serial.print("\* "); break;

}

}

}

Serial.println();

}

Serial.printf("Mine Sensor: %d\n" , sensorValue);

}

**Explanation:**

What the robot has:

1. Wheels to move around
2. An ultrasonic sensor (like a robot's eyes) to detect obstacles
3. A sensor to detect mines

What it does:

Input: The robot reads from its mine sensor and obstacle sensor as it moves around

Process: It systematically explores a 10x10 grid area, moving square by square like mowing a lawn

Output:

Sends a live map via Bluetooth showing where it's been, where it found mines (M), and where obstacles (O) are located

Prints status updates to help monitor its progress

How it works:

* Robot starts at one corner and begins exploring
* At each spot, it checks for mines and obstacles
* If it finds a mine, it marks the location and carefully moves away
* If it hits an obstacle, it goes around it
* It keeps track of everywhere it's been on a virtual map
* Once it's explored the entire area, it returns to where it started
* Throughout the mission, it sends real-time updates showing what it's discovered

Think of it like a smart vacuum cleaner, but instead of cleaning, it's mapping dangerous areas and finding hidden mines while avoiding obstacles.