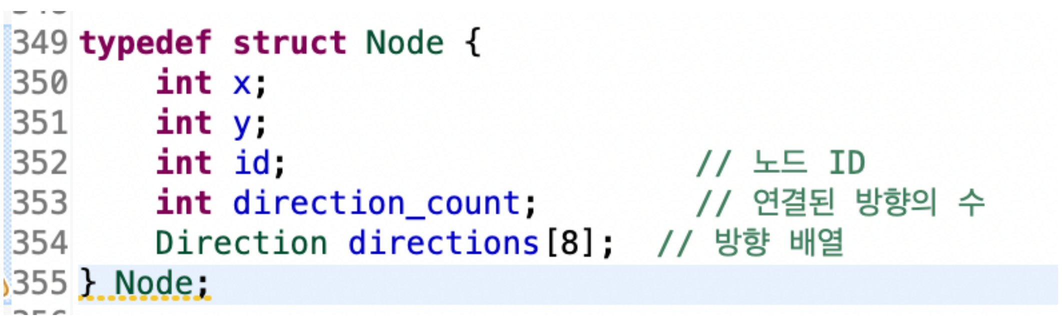
Line Tracer Report

2019060546 주원웅
2019028868 최성현

Variable Explanation

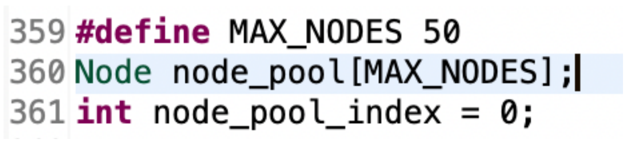


```c
typedef struct Node {
    int x;
    int y;
    int id;                     // 노드 ID
    int direction_count;        // 연결된 방향의 수
    Direction directions[8];  // 방향 배열
} Node;
```

Node Structure:

The node structure represents each vertex in the graph. Each node contains the following information:

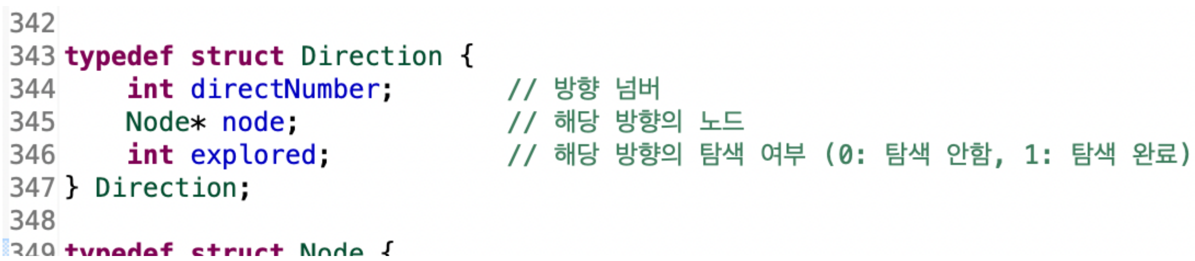
- **Coordinates (x, y)**: The position of the node.
- **ID (id)**: A unique identifier for the node.
- **Direction Count (direction_count)**: The number of edges (directions) connected to the node.
- **Directions Array (directions[8])**: An array holding the edges connected to the node, where each direction is represented as an index from 0 to 7.



```c
#define MAX_NODES 50
Node node_pool[MAX_NODES];
int node_pool_index = 0;
```

Node Pool:

The `node_pool` array is used to allocate memory for nodes statically, addressing heap memory issues. `MAX_NODES` is set to 50, meaning we can store up to 50 nodes. The `node_pool_index` keeps track of the current number of created nodes, incrementing by one with each new node creation.



```c

typedef struct Direction {
    int directNumber;         // 방향 넘버
    Node* node;               // 해당 방향의 노드
    int explored;             // 해당 방향의 탐색 여부 (0: 탐색 안함, 1: 탐색 완료)
} Direction;

typedef struct Node {
```

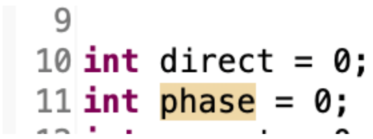
Directions:

A direction in a node represents an edge. Each node can have up to 8 edges, oriented 45 degrees counterclockwise from the north. The direction numbering is as follows:

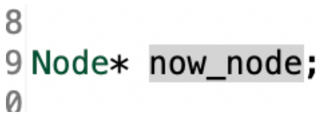
- **0**: North
- **1**: North-West
- **2**: West
- **3**: South-West
- **4**: South
- **5**: South-East
- **6**: East
- **7**: North-East

Each direction contains:

- **Direction Number (directNumber)**: The index (0-7) representing the direction.
- **Node (node)**: The node located in the specified direction.
- **Explored (explored)**: A flag indicating whether the path in this direction has been explored.



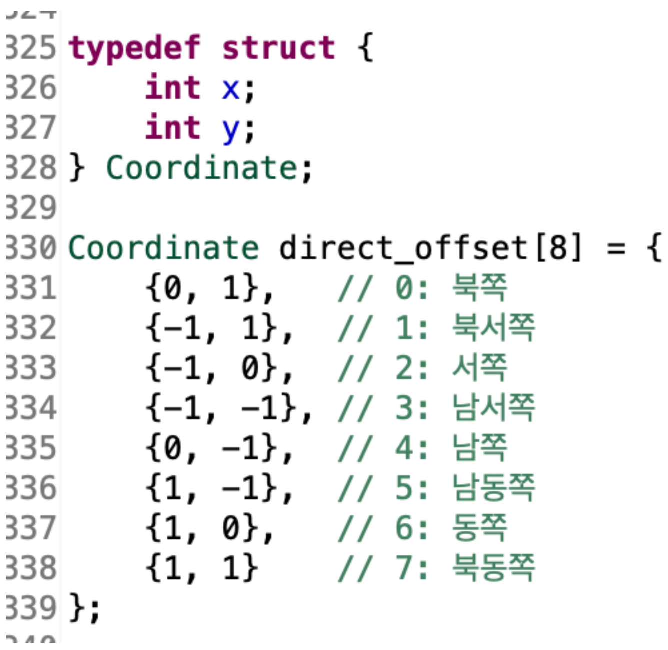
```c

int direct = 0;
int phase = 0;
```



```c

Node* now_node;

```

Current State Variables:

- **Now_node**: The node where the device is currently located.
- **Direct**: The direction the device is currently facing (0-7).
- **Phase**: The current phase of the algorithm (0-3), which dictates the operation being performed.



```c
typedef struct {
    int x;
    int y;
} Coordinate;

Coordinate direct_offset[8] = {
    {0, 1},    // 0: 북쪽
    {-1, 1},   // 1: 북서쪽
    {-1, 0},   // 2: 서쪽
    {-1, -1},  // 3: 남서쪽
    {0, -1},   // 4: 남쪽
    {1, -1},   // 5: 남동쪽
    {1, 0},    // 6: 동쪽
    {1, 1}     // 7: 북동쪽
};
```

Coordinate Structure:

The `Coordinate` structure is used to represent node positions as coordinates. The `direct_offset` array provides the coordinate offsets for each direction index, facilitating the calculation of the coordinates of neighboring nodes.

Algorithm Phases

The algorithm is divided into four phases (phase 0 to 3):

Phase 0: Initial Exploration

Starting from the initial node, the device rotates 45 degrees counterclockwise and moves to the next node if a path exists. To explore all nodes, the device always rotates 45 degrees to the right from the current node, checking for paths and moving to the next node if a path is found. The paths between nodes are connected, and the paths are marked as explored. Phase 0 ends when the device returns to the start node.

Result: All vertices are stored as nodes, and their positions are stored as coordinates. External edges (sides of the octagon) for each node are also stored.

Phase 1: Exploring Remaining Paths

Starting from the initial node, the device rotates 45 degrees clockwise, checking for unexplored paths and moving to the next node if an unexplored path is found. Using the coordinates saved in phase 0, the paths between the start node and destination node are connected and marked as explored. Phase 1 ends when all paths are explored.

Result: All edges are stored, thus storing all map information.

Phase 2: Optimal Path Search

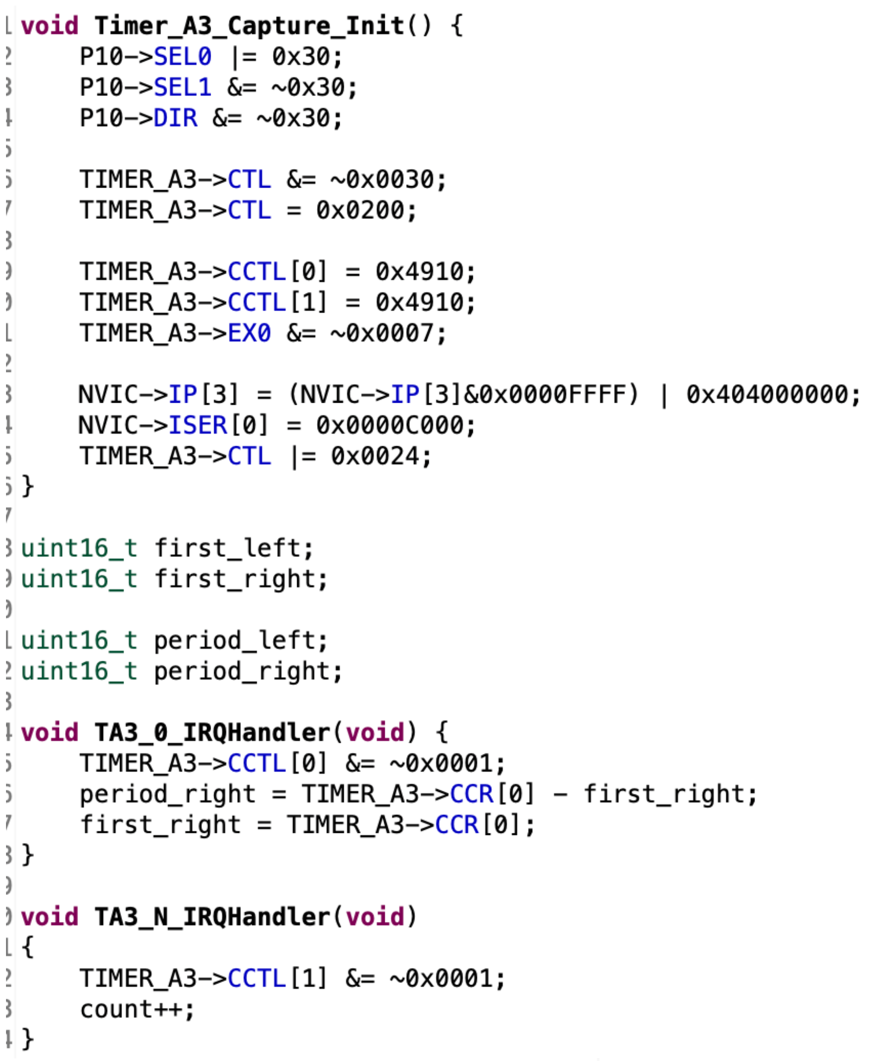
To differentiate from the previous phases, the `go_phase2` function is newly written. This function increases the speed by adding 3000 to the current speed. It uses the `Find_Min_Direct()` function to determine the direction to move. Depending on the direction, the device rotates all at once using `Turn_Right2()` or `Turn_Left2()`. If there are no paths to move to, the `Find_Min_Direct()` function returns a value of 10, indicating that the entire path tracing is complete. Phase 2 then ends, and the process moves to phase 3.

Result: The device visits all edges exactly once in the shortest time possible and returns to the start node.

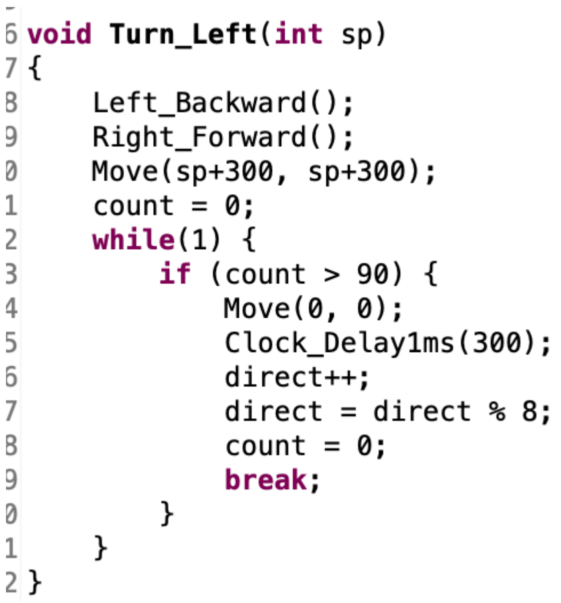
Phase 3: Completion

When all phases are completed, the LED lights up blue to indicate completion.

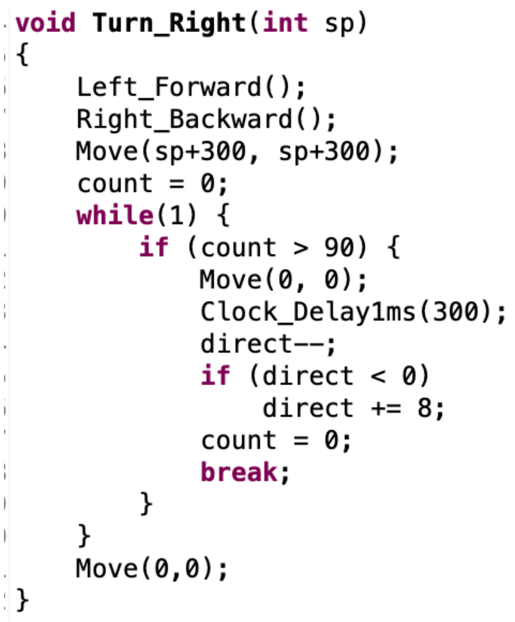
Implement



```c
void Timer_A3_Capture_Init() {
    P10->SEL0 |= 0x30;
    P10->SEL1 &= ~0x30;
    P10->DIR &= ~0x30;

    TIMER_A3->CTL &= ~0x0030;
    TIMER_A3->CTL = 0x0200;

    TIMER_A3->CCTL[0] = 0x4910;
    TIMER_A3->CCTL[1] = 0x4910;
    TIMER_A3->EX0 &= ~0x0007;

    NVIC->IP[3] = (NVIC->IP[3]&0x0000FFFF) | 0x404000000;
    NVIC->ISER[0] = 0x0000C000;
    TIMER_A3->CTL |= 0x0024;
}

uint16_t first_left;
uint16_t first_right;

uint16_t period_left;
uint16_t period_right;

void TA3_0_IRQHandler(void) {
    TIMER_A3->CCTL[0] &= ~0x0001;
    period_right = TIMER_A3->CCR[0] - first_right;
    first_right = TIMER_A3->CCR[0];
}

void TA3_N_IRQHandler(void)
{
    TIMER_A3->CCTL[1] &= ~0x0001;
    count++;
}
```



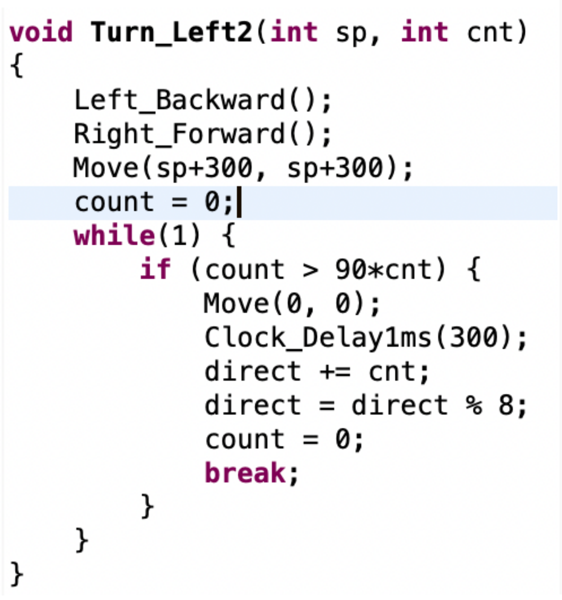
```c
void Turn_Left(int sp)
{
    Left_Backward();
    Right_Forward();
    Move(sp+300, sp+300);
    count = 0;
    while(1) {
        if (count > 90) {
            Move(0, 0);
            Clock_Delay1ms(300);
            direct++;
            direct = direct % 8;
            count = 0;
            break;
        }
    }
}
```



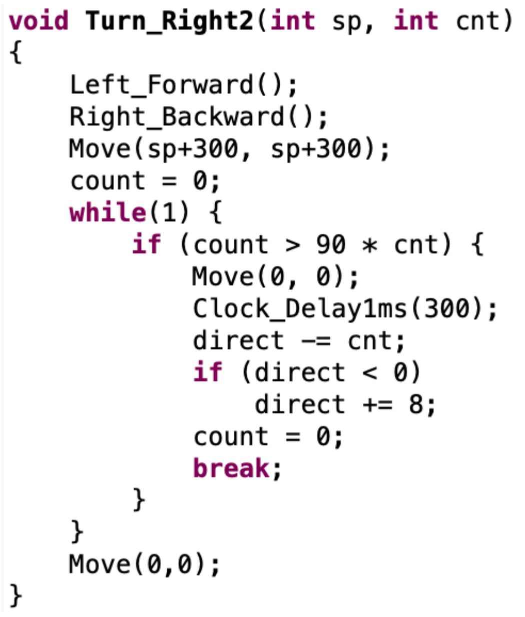
```c
void Turn_Right(int sp)
{
    Left_Forward();
    Right_Backward();
    Move(sp+300, sp+300);
    count = 0;
    while(1) {
        if (count > 90) {
            Move(0, 0);
            Clock_Delay1ms(300);
            direct--;
            if (direct < 0)
                direct += 8;
            count = 0;
            break;
        }
    }
    Move(0,0);
}
```

Interrupt-Based Device Rotation

Using the above code, device rotation through interrupts was implemented. A function that rotates the device by 45 degrees was written using interrupts. This function also updates the direction the device is currently facing (`direct`).



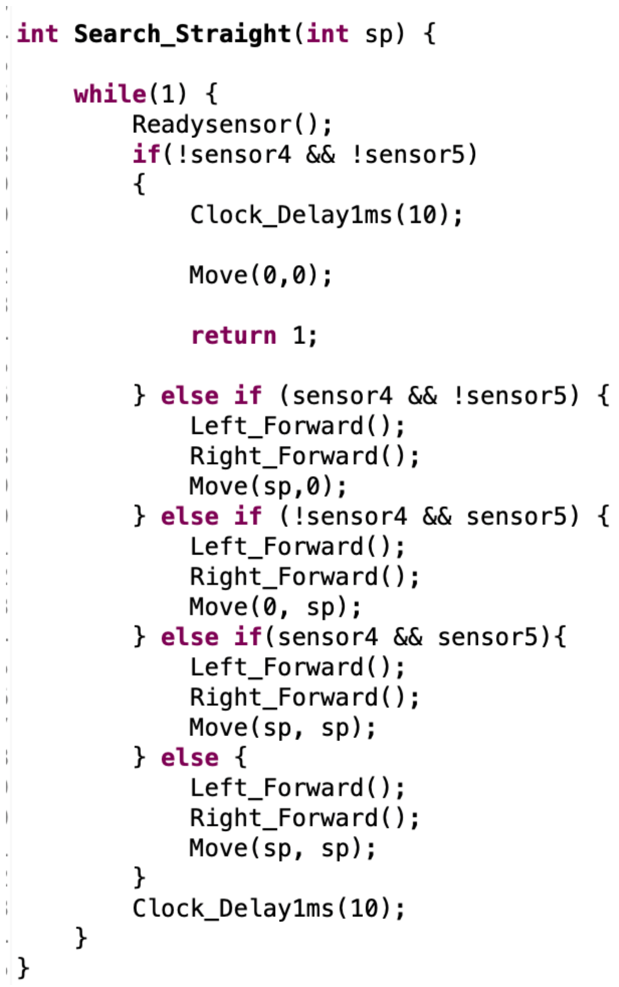
```c
void Turn_Left2(int sp, int cnt)
{
    Left_Backward();
    Right_Forward();
    Move(sp+300, sp+300);
    count = 0;
    while(1) {
        if (count > 90*cnt) {
            Move(0, 0);
            Clock_Delay1ms(300);
            direct += cnt;
            direct = direct % 8;
            count = 0;
            break;
        }
    }
}
```



```c
void Turn_Right2(int sp, int cnt)
{
    Left_Forward();
    Right_Backward();
    Move(sp+300, sp+300);
    count = 0;
    while(1) {
        if (count > 90 * cnt) {
            Move(0, 0);
            Clock_Delay1ms(300);
            direct -= cnt;
            if (direct < 0)
                direct += 8;
            count = 0;
            break;
        }
    }
    Move(0,0);
}
```

Left2 and Right2 Functions

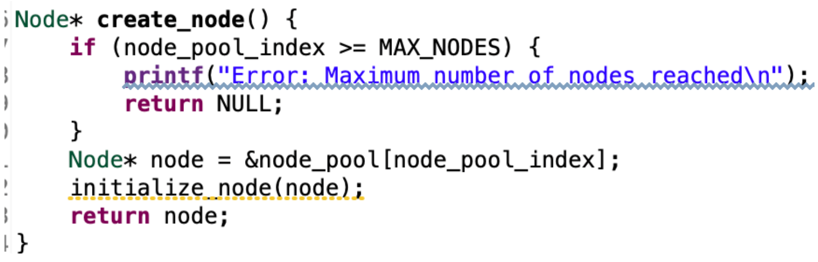
The `left2` and `right2` functions are written to rotate the device to the desired direction at once, rather than 45 degrees at a time, in phase 2. These functions rotate the device by the amount specified by the `cnt` parameter and update the current direction of the device.



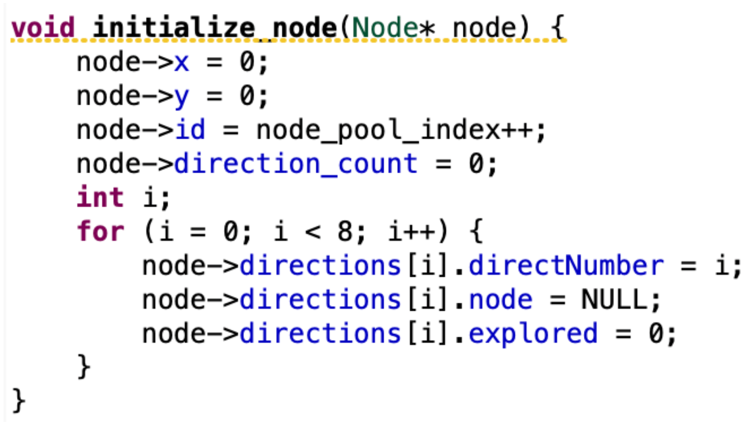
```
int Search_Straight(int sp) {

    while(1) {
        Readysensor();
        if(!sensor4 && !sensor5)
        {
            Clock_Delay1ms(10);

            Move(0,0);

            return 1;

        } else if (sensor4 && !sensor5) {
            Left_Forward();
            Right_Forward();
            Move(sp,0);
        } else if (!sensor4 && sensor5) {
            Left_Forward();
            Right_Forward();
            Move(0, sp);
        } else if(sensor4 && sensor5){
            Left_Forward();
            Right_Forward();
            Move(sp, sp);
        } else {
            Left_Forward();
            Right_Forward();
            Move(sp, sp);
        }
        Clock_Delay1ms(10);
    }
}
```

Straight Movement Function

A function that allows the device to move straight along a path to reach a node was implemented.



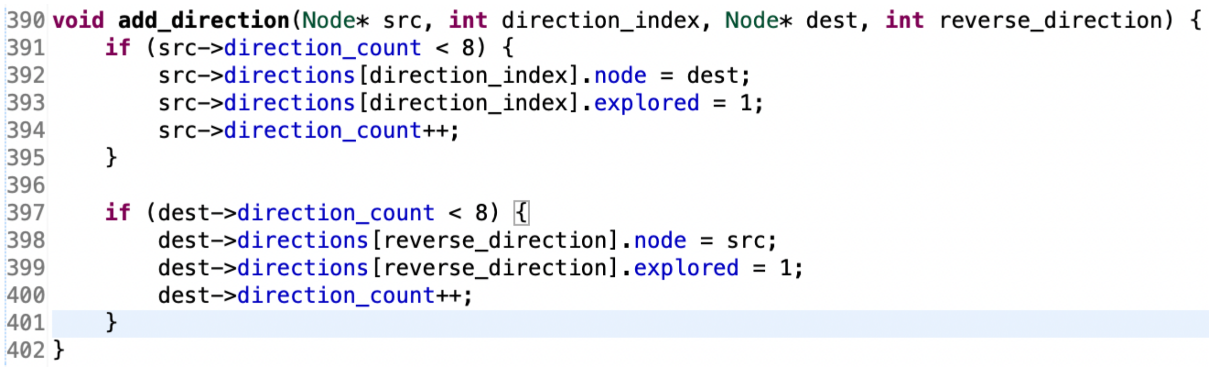
```
Node* create_node() {
    if (node_pool_index >= MAX_NODES) {
        printf("Error: Maximum number of nodes reached\n");
        return NULL;
    }
    Node* node = &node_pool[node_pool_index];
    initialize_node(node);
    return node;
}
```



```
void initialize_node(Node* node) {
    node->x = 0;
    node->y = 0;
    node->id = node_pool_index++;
    node->direction_count = 0;
    int i;
    for (i = 0; i < 8; i++) {
        node->directions[i].directNumber = i;
        node->directions[i].node = NULL;
        node->directions[i].explored = 0;
    }
}
```

Node Creation and Initialization Function

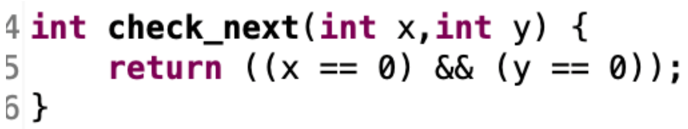
The node creation and initialization function creates a new node and initializes it with default values, setting up the node's state and assigning the necessary initial data.



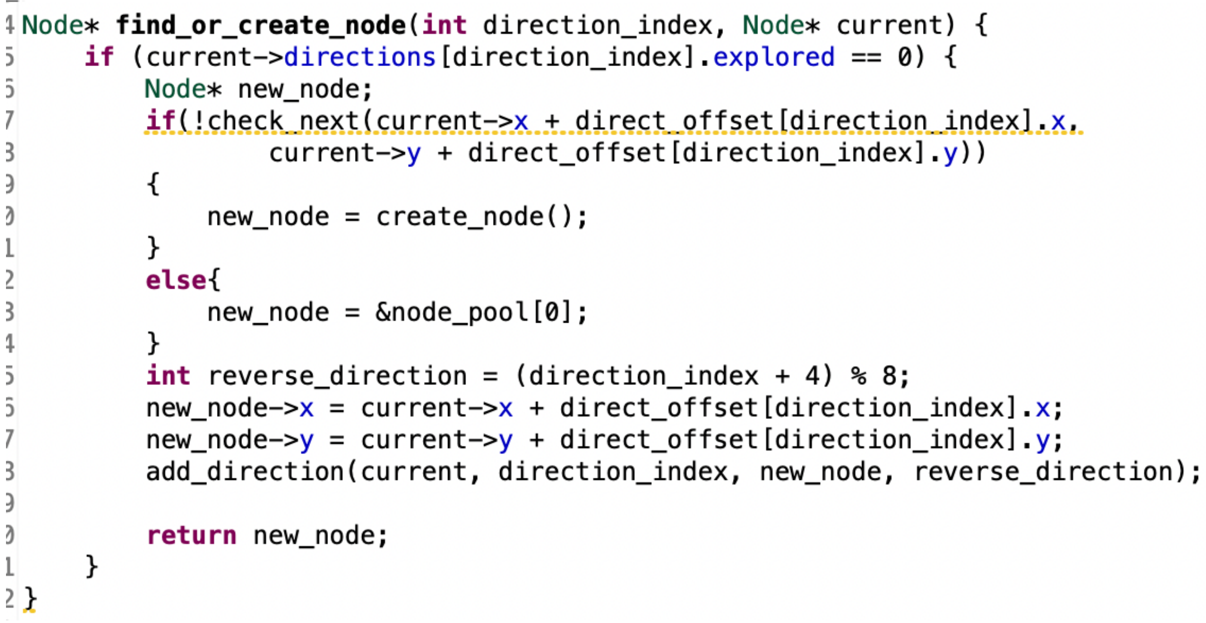
```
void add_direction(Node* src, int direction_index, Node* dest, int reverse_direction) {
    if (src->direction_count < 8) {
        src->directions[direction_index].node = dest;
        src->directions[direction_index].explored = 1;
        src->direction_count++;
    }

    if (dest->direction_count < 8) {
        dest->directions[reverse_direction].node = src;
        dest->directions[reverse_direction].explored = 1;
        dest->direction_count++;
    }
}
```

Node Connection Function

The function that implements the connection between nodes takes `src` (source node) and `dest` (destination node) as parameters. `direction_index` represents the direction of the path from the source node, and `reverse_direction` represents the direction from the destination node. This function connects the nodes and marks the edge as explored.



```c
int check_next(int x,int y) {
    return ((x == 0) && (y == 0));
}
```



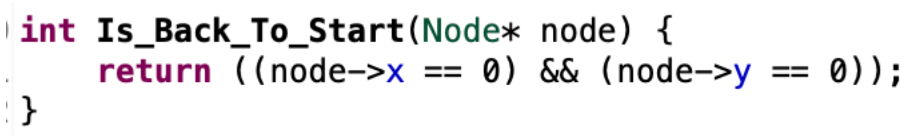
```c
Node* find_or_create_node(int direction_index, Node* current) {
    if (current->directions[direction_index].explored == 0) {
        Node* new_node;
        if(!check_next(current->x + direct_offset[direction_index].x,
                current->y + direct_offset[direction_index].y))
        {
            new_node = create_node();
        }
        else{
            new_node = &node_pool[0];
        }
        int reverse_direction = (direction_index + 4) % 8;
        new_node->x = current->x + direct_offset[direction_index].x;
        new_node->y = current->y + direct_offset[direction_index].y;
        add_direction(current, direction_index, new_node, reverse_direction);

        return new_node;
    }
}
```

Check Next Function

The `check_next` function takes `x` and `y` coordinates as parameters and checks whether these coordinates correspond to the start node (0, 0).

Find or Create Node Function

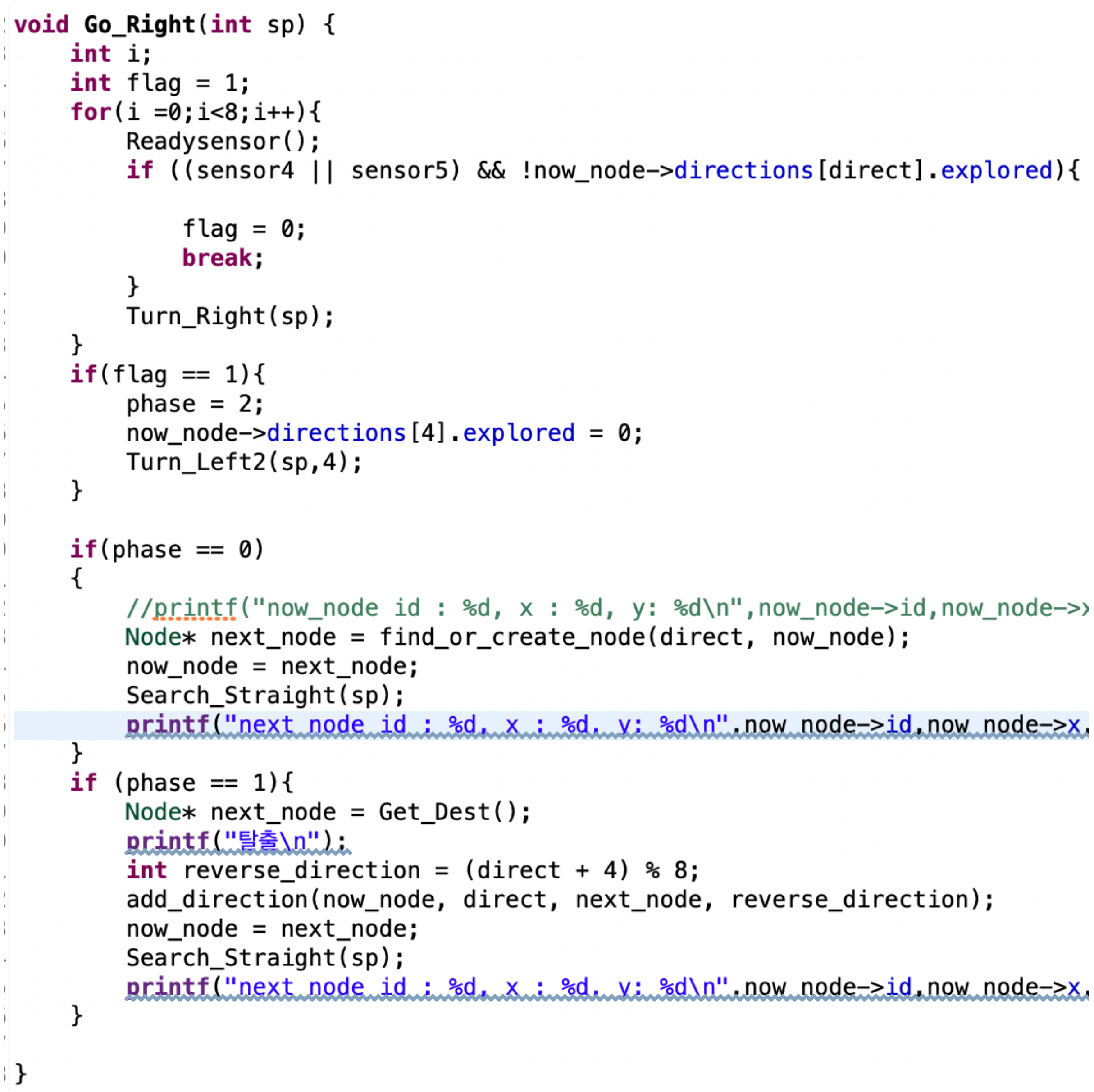
The `find_or_create_node` function takes an index (`direction_index`) representing an existing path from the current node. If the path has not been explored, it creates a new node. If the next node is the start node, it does not create a new node but returns the start node. It then connects the retrieved node with the current node using the `add_direction` function.



```c
int Is_Back_To_Start(Node* node) {
    return ((node->x == 0) && (node->y == 0));
}
```

Is Back to Start Function

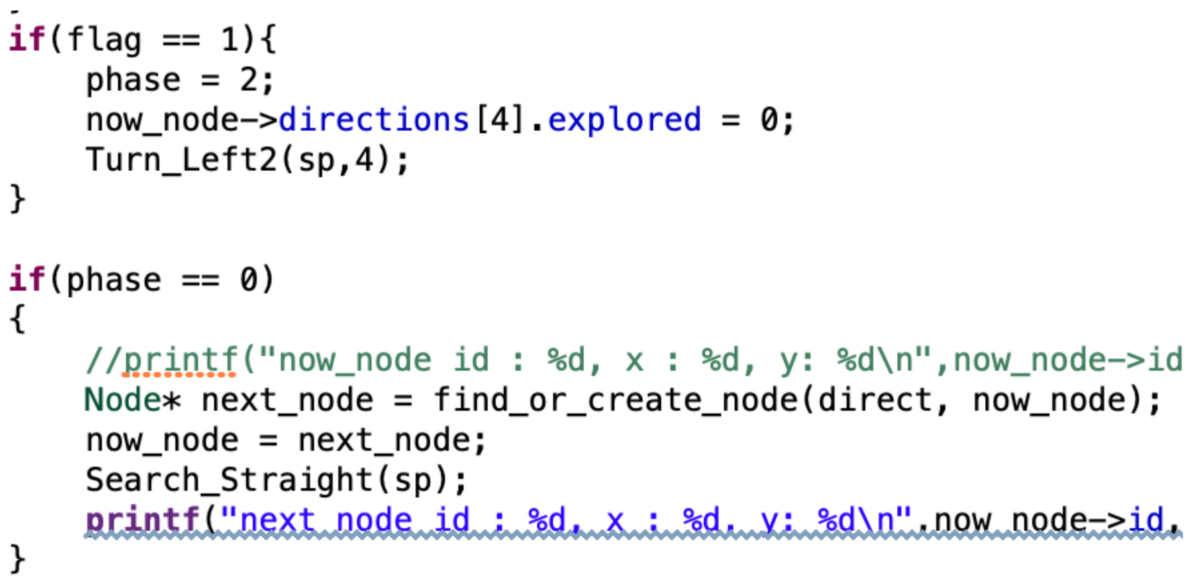
The `is_back_to_start` function takes a node as a parameter and checks if it is the start node.



```
void Go_Right(int sp) {
    int i;
    int flag = 1;
    for(i =0;i<8;i++){
        Readysensor();
        if ((sensor4 || sensor5) && !now_node->directions[direct].explored){

            flag = 0;
            break;
        }
        Turn_Right(sp);
    }
    if(flag == 1){
        phase = 2;
        now_node->directions[4].explored = 0;
        Turn_Left2(sp,4);
    }

    if(phase == 0)
    {
        //printf("now_node id : %d, x : %d, y: %d\n",now_node->id,now_node->x
        Node* next_node = find_or_create_node(direct, now_node);
        now_node = next_node;
        Search_Straight(sp);
        printf("next node id : %d, x : %d, y: %d\n",now_node->id,now_node->x,
    }
    if (phase == 1){
        Node* next_node = Get_Dest();
        printf("탈출\n");
        int reverse_direction = (direct + 4) % 8;
        add_direction(now_node, direct, next_node, reverse_direction);
        now_node = next_node;
        Search_Straight(sp);
        printf("next node id : %d, x : %d, y: %d\n",now_node->id,now_node->x,
    }

}
```

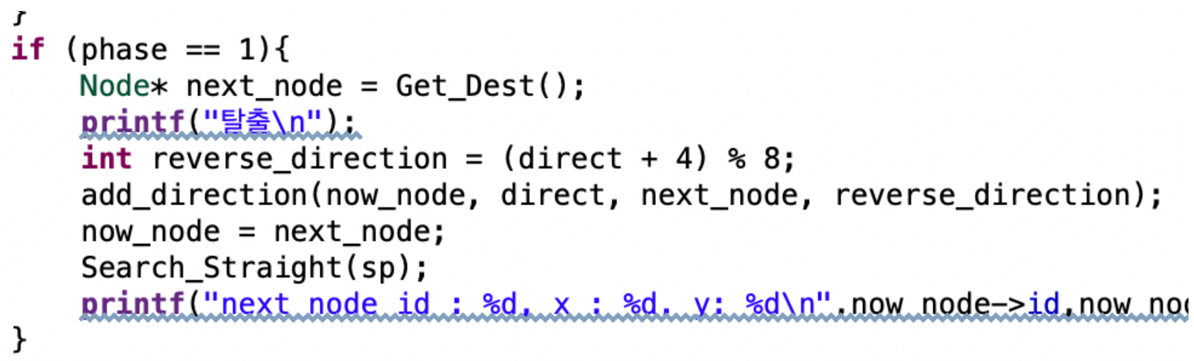
Go_Right Function

The `Go_Right` function rotates the device 45 degrees to the right from the current node, checking if there is a path. If a path exists and has not been explored, the loop exits, and the device moves forward.



```
if(flag == 1){
    phase = 2;
    now_node->directions[4].explored = 0;
    Turn_Left2(sp,4);
}

if(phase == 0)
{
    //printf("now_node id : %d, x : %d, y: %d\n",now_node->id
    Node* next_node = find_or_create_node(direct, now_node);
    now_node = next_node;
    Search_Straight(sp);
    printf("next node id : %d, x : %d, y: %d\n",now_node->id,
}
```

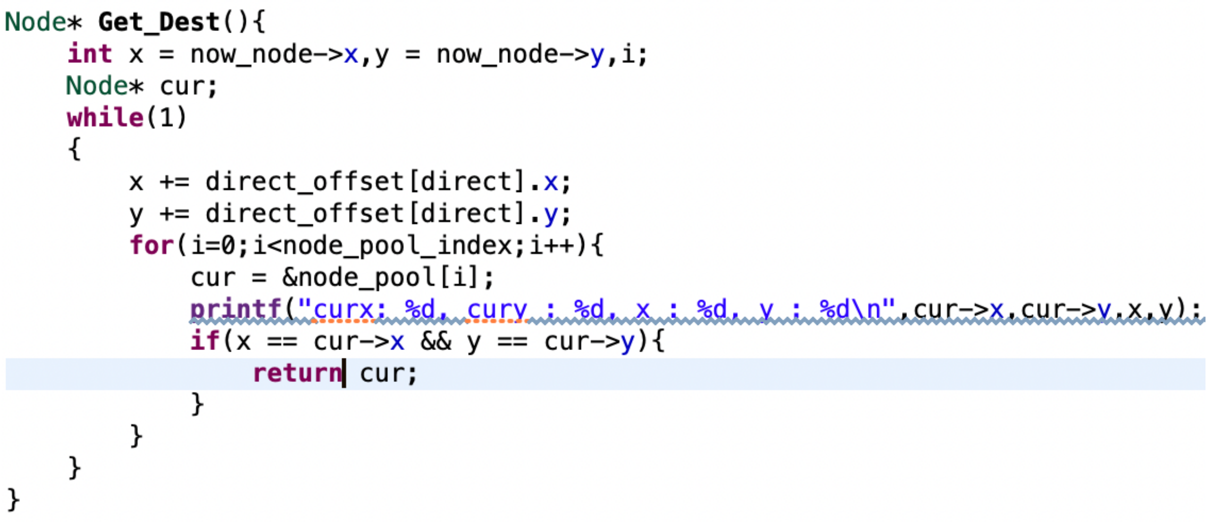
- **Flag**: Indicates that all directions either have no paths or all paths have been explored, meaning the map exploration is complete. In this case, it sets `phase` to 2 to execute the optimal path tracing.
- **Loop Exit and Phase 0**: Creates a new node and moves straight.



```
if (phase == 1){
    Node* next_node = Get_Dest();
    printf("탈출\n");
    int reverse_direction = (direct + 4) % 8;
    add_direction(now_node, direct, next_node, reverse_direction);
    now_node = next_node;
    Search_Straight(sp);
    printf("next node id : %d, x : %d, y: %d\n",now_node->id,now_no
}
```

Loop Exit and Phase 1: Finds the node connected to the path using the `get_dest` function, connects the nodes, and moves straight.

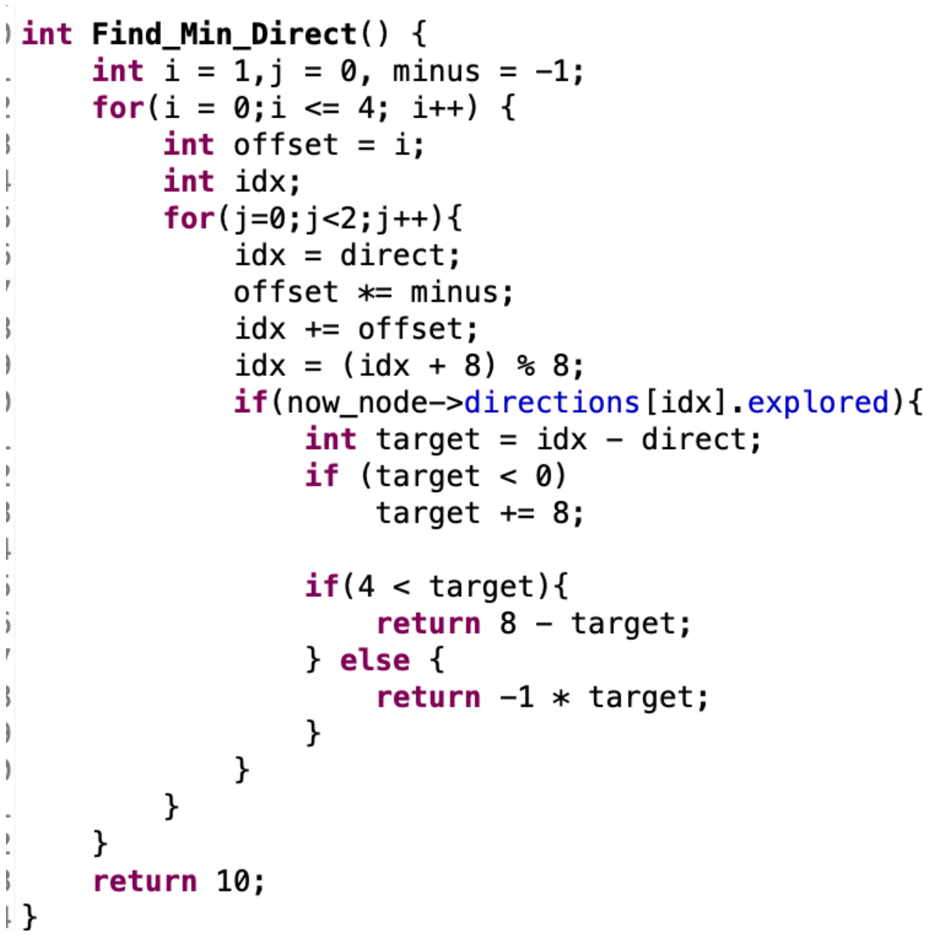
This function explores all paths on the map and creates new nodes or connects existing nodes as the device moves.



```c
Node* Get_Dest(){
    int x = now_node->x,y = now_node->y,i;
    Node* cur;
    while(1)
    {
        x += direct_offset[direct].x;
        y += direct_offset[direct].y;
        for(i=0;i<node_pool_index;i++){
            cur = &node_pool[i];
            printf("curx: %d, cury : %d, x : %d, y : %d\n",cur->x,cur->y,x,y);
            if(x == cur->x && y == cur->y){
                return cur;
            }
        }
    }
}
```

- **Loop Exit and Phase 1**: Finds the node connected to the path using the `get_dest` function, connects the nodes, and moves straight.

This function explores all paths on the map and creates new nodes or connects existing nodes as the device moves.



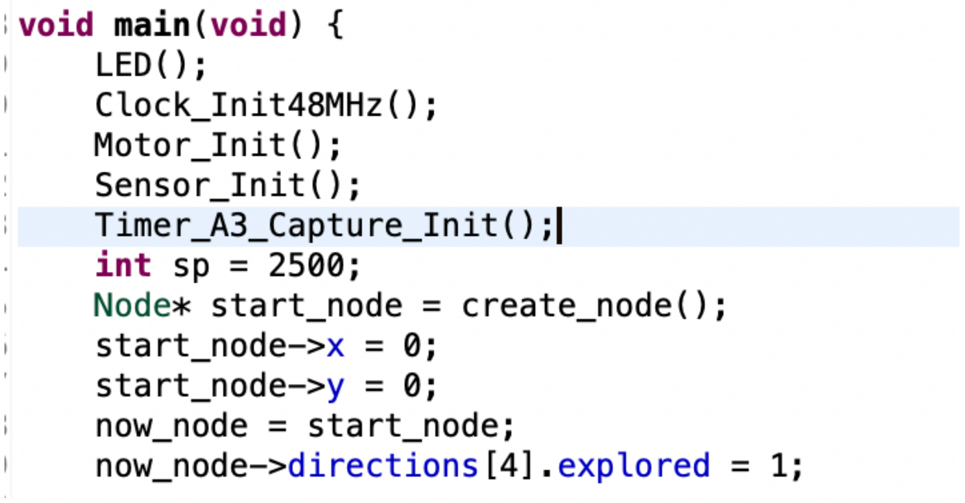
```c
int Find_Min_Direct() {
    int i = 1,j = 0, minus = -1;
    for(i = 0;i <= 4; i++) {
        int offset = i;
        int idx;
        for(j=0;j<2;j++){
            idx = direct;
            offset *= minus;
            idx += offset;
            idx = (idx + 8) % 8;
            if(now_node->directions[idx].explored){
                int target = idx - direct;
                if (target < 0)
                    target += 8;

                if(4 < target){
                    return 8 - target;
                } else {
                    return -1 * target;
                }
            }
        }
    }
    return 10;
}
```

Find_Min_Direct Function

Using `now_node` and `direct`, the `Find_Min_Direct` function returns the direction value of the path with the least rotation among unexplored edges from the current node.

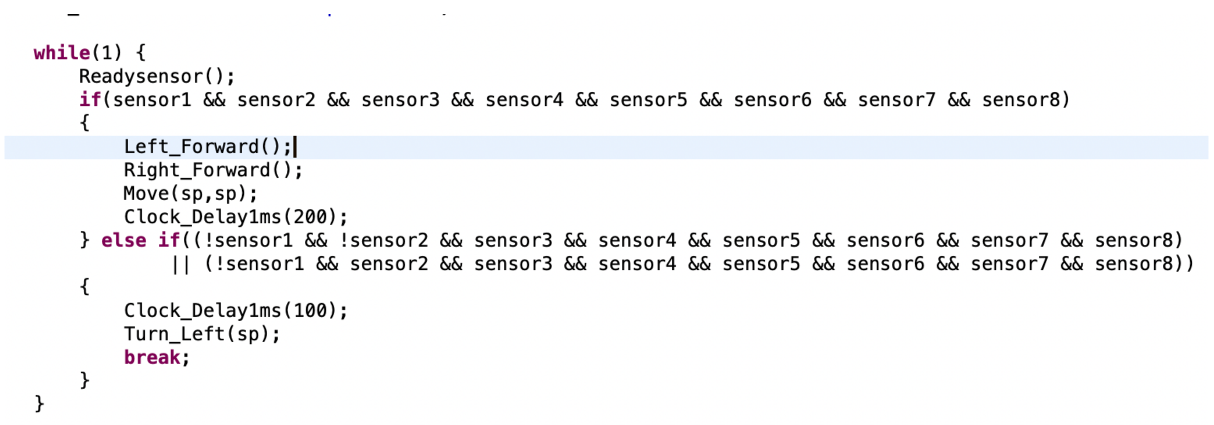
Phase 0: Initial Exploration

Explanation of Code Implementation in `main` Function for Phases 0 to 3



```
void main(void) {
    LED();
    Clock_Init48MHz();
    Motor_Init();
    Sensor_Init();
    Timer_A3_Capture_Init();
    int sp = 2500;
    Node* start_node = create_node();
    start_node->x = 0;
    start_node->y = 0;
    now_node = start_node;
    now_node->directions[4].explored = 1;
```

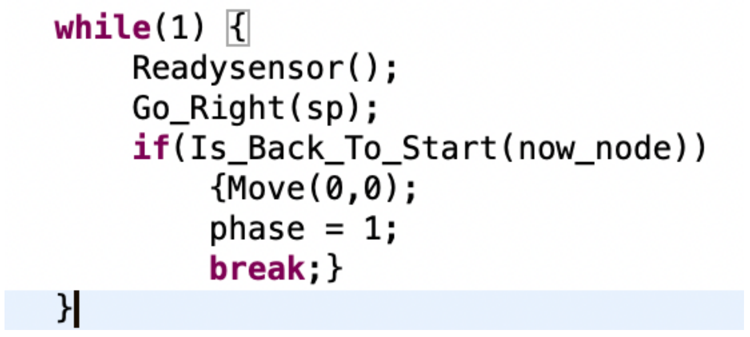
Starting from the initial node, create the start node using the `create_node()` function and set its coordinates to (0, 0). Mark the start node's direction[4] as explored since the map starts from the east (direction = 4).



```
while(1) {
    Readysensor();
    if(sensor1 && sensor2 && sensor3 && sensor4 && sensor5 && sensor6 && sensor7 && sensor8)
    {
        Left_Forward();
        Right_Forward();
        Move(sp,sp);
        Clock_Delay1ms(200);
    } else if((!sensor1 && !sensor2 && sensor3 && sensor4 && sensor5 && sensor6 && sensor7 && sensor8)
            || (!sensor1 && sensor2 && sensor3 && sensor4 && sensor5 && sensor6 && sensor7 && sensor8))
    {
        Clock_Delay1ms(100);
        Turn_Left(sp);
        break;
    }
}
```

Loop:

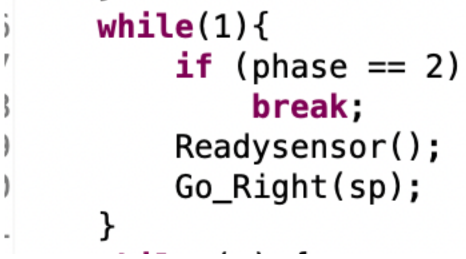
- The `while` loop continuously rotates the device 45 degrees counterclockwise (`Turn_Left`) from the start node, checking for paths.


```
while(1) {
    Readysensor();
    Go_Right(sp);
    if(Is_Back_To_Start(now_node))
        {Move(0,0);
        phase = 1;
        break;}
}
```

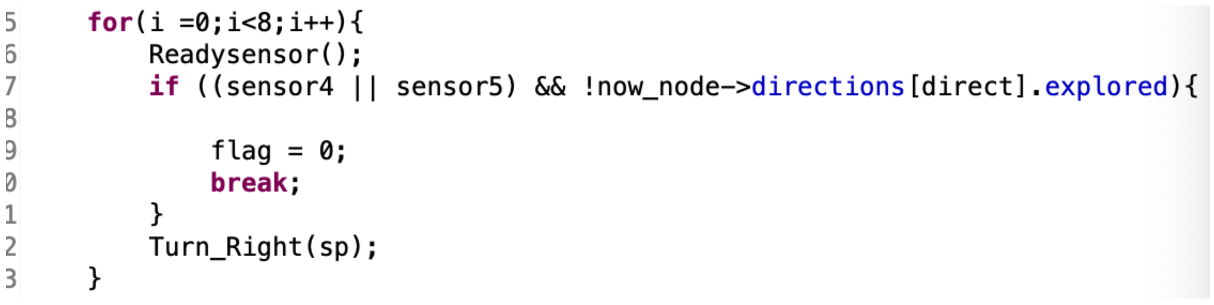
Node Exploration:

- Use the `Go_Right()` function to explore all nodes by rotating 45 degrees to the right from the current node, checking for paths. If a path is found, move to the next node, connect the paths between nodes, and mark them as explored.
- If the device returns to the start node using the `Is_Back_To_Start()` function, phase 0 ends, and phase 1 begins.

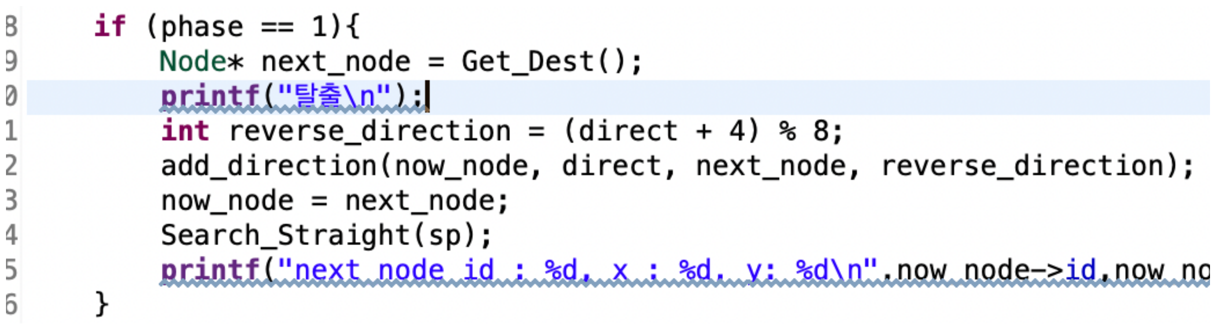
phase 1: Exploring Remaining Paths


```
while(1){
    if (phase == 2)
        break;
    Readysensor();
    Go_Right(sp);
}
```

If the phase is set to 2, the function proceeds to the next phase.

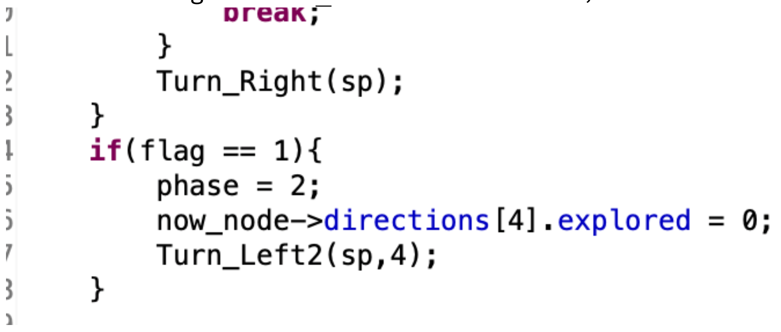

```
for(i =0;i<8;i++){
    Readysensor();
    if ((sensor4 || sensor5) && !now_node->directions[direct].explored){

        flag = 0;
        break;
    }
    Turn_Right(sp);
}
```

Rotate the device 45 degrees to the right to find unexplored paths.



```
    if (phase == 1){
        Node* next_node = Get_Dest();
        printf("탈출\n");
        int reverse_direction = (direct + 4) % 8;
        add_direction(now_node, direct, next_node, reverse_direction);
        now_node = next_node;
        Search_Straight(sp);
        printf("next_node_id : %d, x : %d, y: %d\n",now_node->id,now_no
    }
```

If a path is found, find the node connected to the path using the `get_dest` function, connect the nodes using the `add_direction` function, and move.

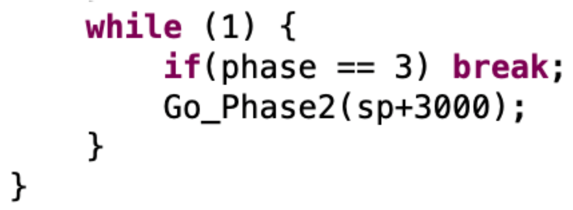


```
                break;
            }
            Turn_Right(sp);
        }
    if(flag == 1){
        phase = 2;
        now_node->directions[4].explored = 0;
        Turn_Left2(sp,4);
    }
```

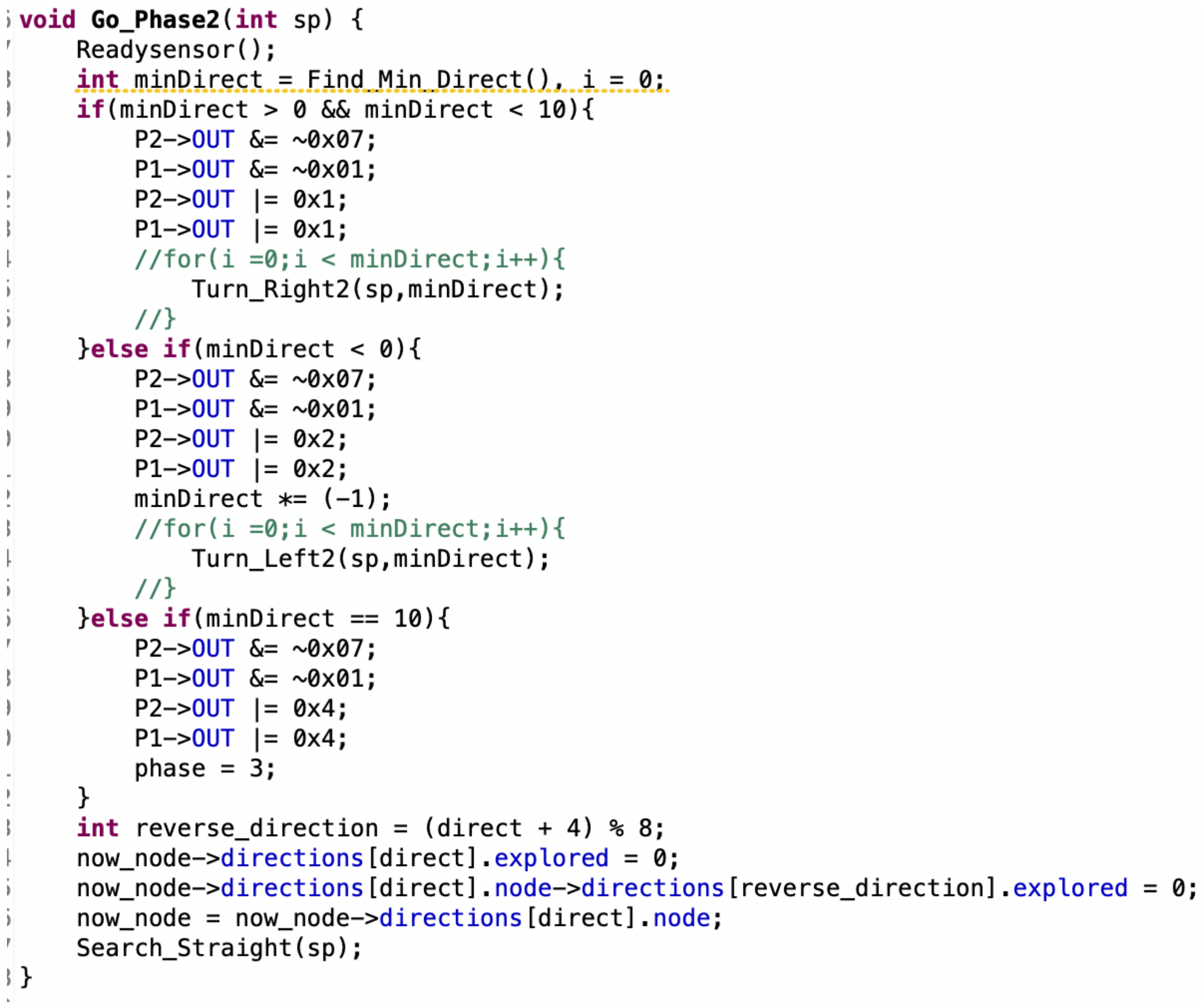
- Repeat this process until all paths are explored (Flag is set to 1).

Result: All edges are stored, thus storing all map information.

Phase 2: Optimal Path Search



```
    while (1) {
        if(phase == 3) break;
        Go_Phase2(sp+3000);
    }
}
```



```c
void Go_Phase2(int sp) {
    Readysensor();
    int minDirect = Find_Min_Direct(), i = 0;
    if(minDirect > 0 && minDirect < 10){
        P2->OUT &= ~0x07;
        P1->OUT &= ~0x01;
        P2->OUT |= 0x1;
        P1->OUT |= 0x1;
        //for(i =0;i < minDirect;i++){
            Turn_Right2(sp,minDirect);
        //}
    }else if(minDirect < 0){
        P2->OUT &= ~0x07;
        P1->OUT &= ~0x01;
        P2->OUT |= 0x2;
        P1->OUT |= 0x2;
        minDirect *= (-1);
        //for(i =0;i < minDirect;i++){
            Turn_Left2(sp,minDirect);
        //}
    }else if(minDirect == 10){
        P2->OUT &= ~0x07;
        P1->OUT &= ~0x01;
        P2->OUT |= 0x4;
        P1->OUT |= 0x4;
        phase = 3;
    }
    int reverse_direction = (direct + 4) % 8;
    now_node->directions[direct].explored = 0;
    now_node->directions[direct].node->directions[reverse_direction].explored = 0;
    now_node = now_node->directions[direct].node;
    Search_Straight(sp);
}
```

Phase 2: Optimal Path Search

To differentiate from phases 0 and 1, the `go_phase2` function was newly written. This function increases the speed by adding 3000 to the current speed. It uses the `Find_Min_Direct()` function to determine the direction to move. Depending on the direction, the device rotates all at once using `Turn_Right2()` or `Turn_Left2()`. If there are no paths to move to, the `Find_Min_Direct()` function returns a value of 10, indicating that the entire path tracing is complete. Phase 2 then ends, and the process moves to phase 3.

Result: The device visits all edges exactly once in the shortest time possible and returns to the start node.

Phase 3: Completion

When all phases are completed, the LED lights up blue to indicate completion.