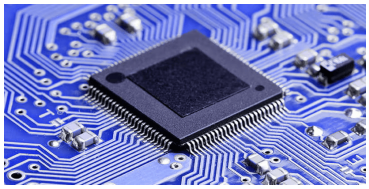


# Memory-Constrained Path Planner for RISC-V Microcontroller

## End Evaluation Report

**IITI Summer Of Code**

**Domain: Electronics**

**Authors: Avni Jharware, Vaibhav Sharma**

The project presents a path planner for an autonomous robot with strict memory constraints, blending RISC-V innovation with embedded software efficiency. It features a compact navigation stack in C, simulated on RIPES, using memory-mapped I/O, efficient path algorithms, and compressed storage for advanced embedded engineering education.

# Acknowledgements

We would like to express our sincere gratitude to all those who have supported and guided us throughout the completion of this project.

First and foremost, we extend our heartfelt thanks to our mentor, Advay kunte for his invaluable guidance, encouragement, and expert advice. His insights and support have been instrumental in overcoming the technical challenges presented by this project.

Our sincere appreciation goes to the Science and Technology Council of IIT Indore and the IITI Summer of Code program for providing the opportunity, educational materials, and a collaborative environment that fostered learning and development.

Lastly, we are grateful to our family and friends for their continued encouragement and patience throughout the duration of this project.

Thank you for your unwavering support.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Modern embedded systems—especially in robotics and automation—demand algorithms and implementations that are both efficient and resource-aware. Unlike desktop or server environments, microcontrollers such as RISC-V based chips offer limited instruction memory, constrained RAM, and no operating system for resource management. In such contexts, even basic tasks like pathfinding and navigation become algorithmically challenging.

Efficient utilization of memory and processor cycles is essential not only for cost-effective designs, but also for achieving real-time guarantees and high system reliability in applications like line-following robots, warehouse automation, and micro-drones. The open-source and modular RISC-V architecture has emerged as an ideal platform for teaching and experimenting with such problems thanks to its simplicity, accessibility, and steadily growing ecosystem.

## 1.2 Project Objectives

The objective of this project is to design and implement a **memory-constrained path planner** in pure C, suitable for a RISC-V RV32I microcontroller system and validated using the RIPES processor simulator. The planner enables a virtual autonomous robot to find and follow the shortest path from a specified start to a goal position with pre-encoded obstacles.

Key goals include:

- Developing an efficient embedded navigation software stack that operates under strict memory and instruction constraints.
- Employing a standard, well-studied pathfinding algorithm—Breadth-First Search (BFS)—tailored for flat arrays and minimum RAM.
- Implementing compressed path storage (2 bits per movement), and a simple FSM (finite state machine) for movement and sensor-based correction.
- Simulating all hardware interactions (UART input, IR sensors, motor commands) with memory-mapped registers matching those typically found in real microcontroller and robotics setups.

- Demonstrating system correctness and constraint compliance within the RIPES simulator, including evidence of low code and memory footprint.

## 1.3 Problem Statement and Constraints

This project specifically addresses the following challenge:

*"Design a navigation software stack for a hypothetical autonomous robot navigating a* $16 \times 16$ *grid environment with static obstacles, using only static data allocation and occupying no more than 512 instructions and 256 bytes of data memory on a RISC-V microcontroller, with the entire stack programmed in C and executed on the RIPES simulator."*

The robot must:

- Receive start and end coordinates via memory-mapped UART
- Calculate the shortest path
- Encode the path in highly compressed form
- Simulate movement step-by-step, checking alignment each step using a simulated IR sensor input, and correct deviations if detected.

## 1.4 Report Organization

This report is structured as follows:

- **Chapter 2:** Presents the node map structure and its role in the software system.
- **Chapter 3:** Explains the design and implementation of the data structures and navigation algorithms, with focus on resource constraints.
- **Chapter 4:** Details validation, experimental results, and compliance with the project's performance and resource limits.
- Appended materials include test scenarios, and references.

The remainder of this report will show how careful system design, algorithm selection, and C programming best-practices yield a solution that is efficient, robust, and directly applicable to real-world embedded applications in constrained hardware environments.

# Chapter 2

# The Environment Map

## 2.1 Node Layout and Adjacency

As shown in Figure 2.1, the robot navigates a graph-style map with 32 nodes, each representing an intersection or bifurcation point. This visual layout matches precisely the array indices and node connections used in the `init_graph()` C function.

## 2.2 Mapping in Code

Each node in the map is represented as a fixed index (0 to 31), and direct connections (edges) are established through a statically-allocated array `arr[32][4]`. For example:

- Node 0 connects up to node 10, right to node 1, down to 6 (left: none).
- Node 1 connects up to node 11, left to node 0, down to node 2 (right: none).

Obstacles are represented by invalid links (marked 255).

## 2.3 Grid Storage Format

The grid storage in this project employs a **node-based graph representation** rather than a traditional 2D occupancy grid. This approach is specifically optimized for the memory constraints of the RISC-V microcontroller while maintaining efficient pathfinding capabilities.

### 2.3.1 Design Philosophy

Instead of storing a complete 16×16 pixel grid (which would require 256 bits or 32 bytes for occupancy data), the system uses a **sparse graph representation** with 32 strategically placed nodes. Each node represents a key decision point or intersection in the environment, significantly reducing memory requirements while preserving essential connectivity information.

### 2.3.2 Adjacency Matrix Implementation

The core of the grid storage is a **static adjacency matrix** implemented as a 2D array:

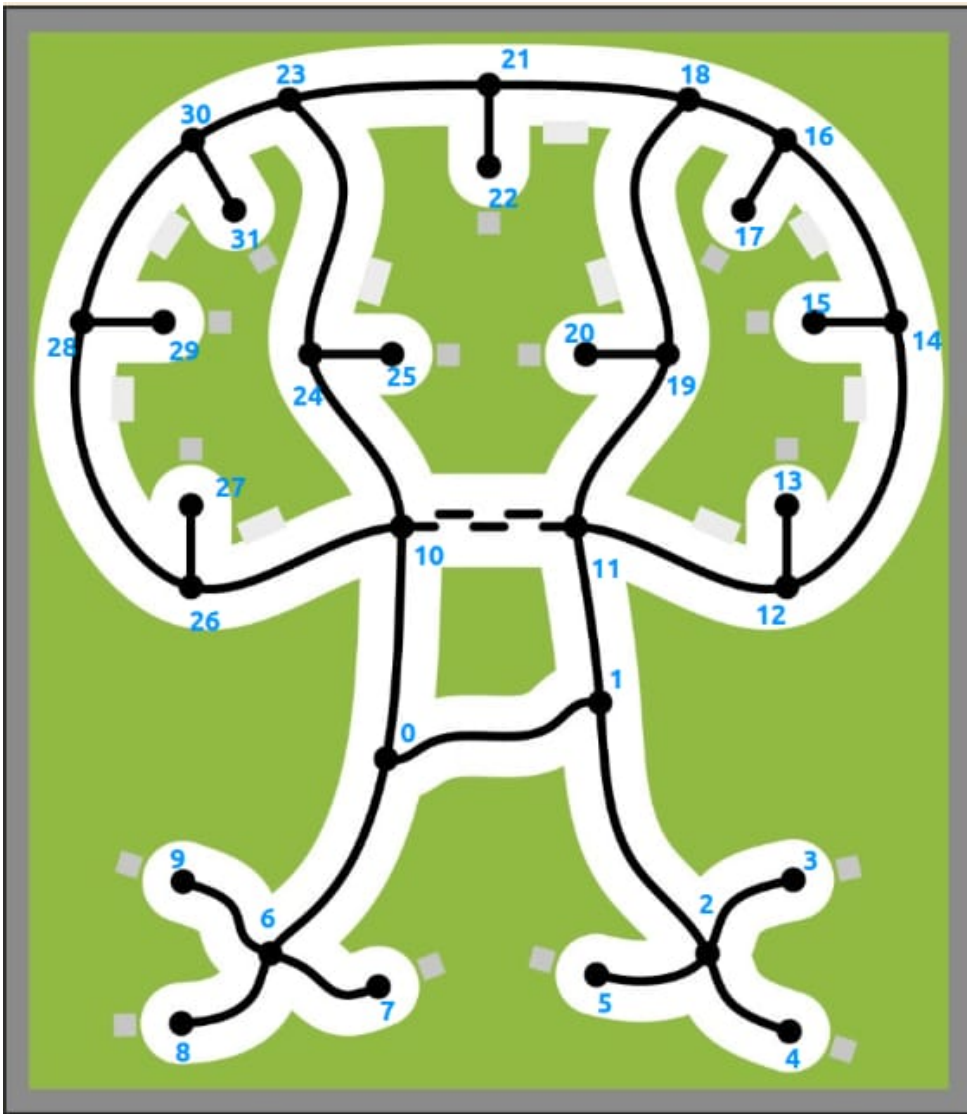



Figure 2.1: Node-based map used by the robot for navigation. Black dots (numbered 0–31) are node indices, blue numbers are node labels, and black lines represent valid paths.

```
#define NODES 32
#define DIRS 4
#define INVALID 255

const uint8_t initial_graph_data[NODES][DIRS] = {
    /* Node 00 */  {10, 1, 6, INVALID},      // UP, RIGHT, DOWN, LEFT
    /* Node 01 */  {11, INVALID, 2, 0},      // INVALID means no
        connection
    /* Node 02 */  {1, 3, 4, 5},             // Node 2 connects in all
        directions
    // ... (continued for all 32 nodes)
};
```

Listing 2.1: Graph Storage Structure

### 2.3.3 Memory Layout and Organization

The adjacency data is stored in a contiguous memory block with the following structure:

- **Base Address:** `0x00011900` (RAM_START_BASE)
- **Array Dimensions:** 32 nodes × 4 directions = 128 bytes
- **Direction Encoding:**
  - Index 0: UP connection (North)
  - Index 1: RIGHT connection (East)
  - Index 2: DOWN connection (South)
  - Index 3: LEFT connection (West)
- **Invalid Connections:** Marked with value 255 (INVALID)

### 2.3.4 Access Pattern and Efficiency

Node connectivity is accessed using a simple 2D array indexing scheme:

```
// Access the RIGHT connection from node 0
uint8_t next_node = arr[0][DIR_RIGHT];  // Returns node 1
if (next_node != INVALID) {
    // Valid path exists to node 1
}
```

Listing 2.2: Grid Access Example

### 2.3.5 Memory Efficiency Analysis

| Component | Size (bytes) | Purpose |
|---|---|---|
| Adjacency Matrix | 128 | Node connectivity data |
| Visited Bitmap | 4 | BFS traversal state |
| Parent Array | 32 | Path reconstruction |
| Queue/Path Array | 32 | BFS queue & path storage |
| Control Variables | 8 | FSM and algorithm state |
| Packed Instructions | 16 | Compressed movement commands |
| Dynamic Path Data | 12 | Runtime path execution |
| **Total** | **232** | **90.6% of 256-byte limit** |

Table 2.1: Complete memory utilization breakdown for grid storage and pathfinding.

### 2.3.6 Advantages of Node-Based Representation

- **Memory Efficiency:** Uses only 128 bytes for connectivity vs. 256+ bytes for full grid
- **Sparse Representation:** Eliminates storage of empty/obstacle regions

- **Fast Access:** Direct O(1) lookup for node connections
- **Scalability:** Easy to modify connectivity without restructuring data
- **Pathfinding Optimization:** BFS operates directly on meaningful waypoints

### 2.3.7 Initialization and Runtime Behavior

The graph is initialized once at startup using the `init_graph()` function, which copies the static graph data into volatile memory-mapped locations:

```
void init_graph() {
    for (int i = 0; i < NODES; i++) {
        for (int j = 0; j < DIRS; j++) {
            arr[i][j] = initial_graph_data[i][j];
        }
    }
}
```

Listing 2.3: Graph Initialization

This approach ensures that the pathfinding algorithm operates on a consistent, memory-efficient representation while maintaining the flexibility to handle dynamic obstacles through runtime modifications to the adjacency matrix.

# Chapter 3

# Code Walkthrough

This chapter delves into the step-by-step logic of our navigation stack, illustrating how each algorithmic component is tailored for memory-constrained, real-time operation on an RV32I microcontroller. We emphasize design decisions, complexity trade-offs, and the synergy between code structure and hardware mapping. Placeholders for illustrative figures are included; replace with your actual diagrams and screenshots.

## 3.1 Breadth-First Search (BFS) Pathfinding

### 3.1.1 Why BFS?

The implementation of BFS was a decision of critical importance for this project. A consideration for choosing a much efficient path planning algorithm like A* was considered but it was discarded due to some important drawbacks that were covered by BFS. A* is computationally intensive, and using that for a simple map with 32 unweighted nodes would have been overkill. A* relies on complex heuristic functions to guide its search and is really helpful in weighted graphs. However, since the 32-node graph provided is unweighted, we tend to use the easily implementable BFS algorithm. Additionally, due to the nature of the BFS algorithm, it provides the shortest path each time consistently and with the same time complexity. This makes this algorithm robust and easy to use for beginners. BFS only requires a simple fixed-size queue for its operation, unlike storing cost and heuristic data that A* would require. The efficiency of this algorithm in small, unweighted graphs makes it more inclined towards being well within the project's 256-byte limit and the strict 512-instruction budget. Thus, BFS was the only viable choice, prioritizing simplicity and resource conservation without compromising accuracy.

### 3.1.2 Overview

Breadth-First Search (BFS) is employed to guarantee the shortest path in an unweighted graph. Our "graph" comprises 32 nodes representing critical waypoints in the environment, connected according to static obstacle-free links.

### 3.1.3 Data Structure Design

- **Adjacency Array** (`arr[32][4]`): A 2D table of 128 bytes mapping each node to up to four neighbors. Invalid directions marked `255`.

- **Visited Bitmap** (`uint32_t visited`): A single 32-bit word tracks visited nodes with bitwise operations, saving 31 bytes compared to a 32-byte boolean array.
- **Parent Array** (`parent[32]`): Records the predecessor for each node, enabling path backtracking.
- **Queue** (`queue[32]`, `front`, `rear`): A circular buffer, implementing BFS frontier, without dynamic memory.

### 3.1.4 Algorithm Flow

1. **Initialization**

```
visited = 0;
front = rear = 0;
for (i = 0; i < NODES; i++) parent[i] = INVALID;
enqueue(start);
mark_visited(start);
```

2. **Level-by-Level Expansion**

```
while (front != rear) {
    uint8_t curr = dequeue();
    for (uint8_t d = 0; d < DIRS; d++) {
        uint8_t nxt = arr[curr][d];
        if (nxt != INVALID && !is_visited(nxt)) {
            mark_visited(nxt);
            parent[nxt] = curr;
            enqueue(nxt);
        }
    }
}
```

3. **Termination** BFS stops when the queue empties or when the `parent[target]` is set—indicating the goal was reached.

### 3.1.5 Complexity and Memory Considerations

- *Time Complexity:* $O(V + E)$, with $V = 32$ nodes, $E \approx 4V$, thus negligible on microcontroller.
- *Space Complexity:* 128 bytes for adjacency, 4 bytes for visited, 32 for parent, 32 for queue = 196 bytes total for BFS structures.
- *Optimization:* Bitwise visitation reduces memory and branch overhead.

## 3.2 Path Reconstruction and Compression

### 3.2.1 Backtracking via Parent Array

Once BFS completes, the path is recovered by tracing `parent[]` pointers:

| | | | | | |
|---|---|---|---|---|---|
| 0x000119c8 | 285736960 | 0 | 0 | 8 | 17 |
| 0x000119c4 | 0 | 0 | 0 | 0 | 0 |
| 0x000119c0 | 286199839 | 31 | 16 | 15 | 17 |
| 0x000119bc | 371002893 | 13 | 14 | 29 | 22 |
| 0x000119b8 | 336731669 | 21 | 30 | 18 | 20 |
| 0x000119b4 | 84149020 | 28 | 3 | 4 | 5 |
| 0x000119b0 | 453776153 | 25 | 19 | 12 | 27 |
| 0x000119ac | 386013707 | 11 | 26 | 2 | 23 |
| 0x000119a8 | 402721289 | 9 | 10 | 1 | 24 |
| 0x000119a4 | 117442056 | 8 | 6 | 0 | 7 |
| 0x000119a0 | 504831002 | 26 | 28 | 23 | 30 |
| 0x0001199c | 436869130 | 10 | 24 | 10 | 26 |
| 0x00011998 | 404035347 | 19 | 23 | 21 | 24 |
| 0x00011994 | 185798674 | 18 | 16 | 19 | 11 |
| 0x00011990 | 235670539 | 11 | 12 | 12 | 14 |
| 0x0001198c | 167773951 | 255 | 6 | 0 | 10 |
| 0x00011988 | 101188098 | 2 | 2 | 8 | 6 |
| 0x00011984 | 33619974 | 6 | 0 | 1 | 2 |
| 0x00011980 | -1 | 255 | 255 | 255 | 255 |

Figure 3.1: BFS frontier expansion.

```
uint8_t temp[NODES], len = 0, cur = target;
while (cur != INVALID) {
    temp[len++] = cur;
    if (cur == start) break;
    cur = parent[cur];
}
path_len = len;
for (i = 0; i < len; i++) {
    path[i] = temp[len - 1 - i];
}
```

### 3.2.2 2-Bit Command Encoding

To minimize RAM footprint, each movement (Forward, Right, Left, Around) is encoded in **2 bits**:

- `CMD_FORWARD = 0`, `CMD_RIGHT = 1`, `CMD_LEFT = 2`, `CMD_AROUND = 3`.
- Commands packed into a byte array (`commands[]`) of 16 bytes allows up to 64 instructions.

```
void encode_instruction(uint8_t* buf, uint idx, uint8_t cmd) {
    uint byte = idx >> 2;
    uint shift = (idx & 0x3) << 1;
    buf[byte] = (buf[byte] & ~(0x3 << shift)) | (cmd << shift);
}
```

## 3.3 Finite State Machine (FSM) Control

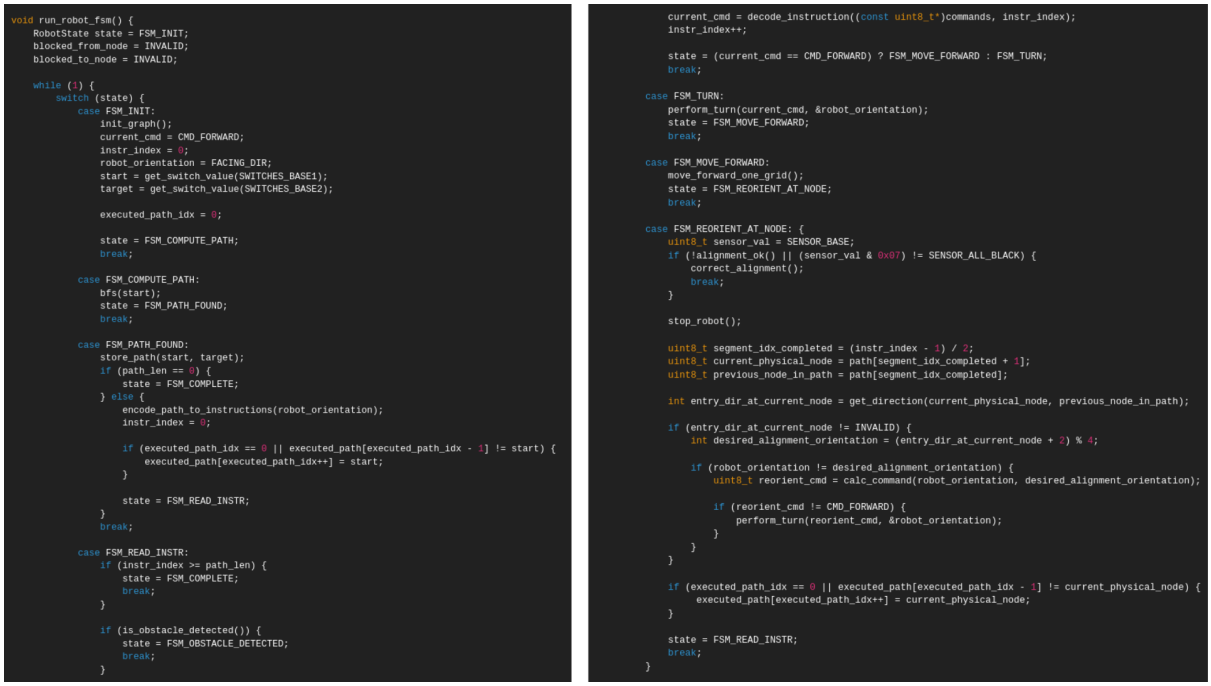
### 3.3.1 State Definitions

- `FSM_INIT`: System and memory setup, UART input of `start` and `target`.
- `FSM_COMPUTE_PATH`: Invoke BFS.
- `FSM_PATH_FOUND`: Reconstruct and encode path.
- `FSM_READ_INSTR`: Fetch next command.
- `FSM_TURN / FSM_MOVE_FORWARD`: Actuate motors in small, predictable bursts.
- `FSM_REORIENT_AT_NODE`: Alignment correction using IR sensors.
- `FSM_OBSTACLE_DETECTED`: Dynamic re-planning upon obstacle detection.
- `FSM_COMPLETE`: Halt and reset.

### 3.3.2 Transition Logic

The FSM's "deterministic state transitions" mean that for any given state and any given input or condition, the robot's next state is always predictable and unambiguous. This prevents unpredictable or erratic behavior. The code achieves this by structuring the main control loop with a `switch(state)` statement. Inside each `case` block, the logic is kept as simple as possible, often with a single conditional `if` statement to decide the next state. By limiting each iteration to "at most one conditional check," the design minimizes complex branching in the compiled machine code. This is vital for a resource-constrained RISC-V microcontroller because it reduces the number of instructions needed for control flow and helps to avoid costly pipeline stalls caused by branch misprediction, ensuring a fast and efficient execution that stays well within your strict instruction count budget.

There is an inserted image of the FSM logic (except features like `FSM_OBSTACLE_DETECTED` and `FSM_COMPLETE`) which demonstrates the flow of instructions inside the robot's processor.



```c
void run_robot_fsm() {
    RobotState state = FSM_INIT;
    blocked_from_node = INVALID;
    blocked_to_node = INVALID;

    while (1) {
        switch (state) {
            case FSM_INIT:
                init_graph();
                current_cmd = CMD_FORWARD;
                instr_index = 0;
                robot_orientation = FACING_DIR;
                start = get_switch_value(SWITCHES_BASE1);
                target = get_switch_value(SWITCHES_BASE2);

                executed_path_idx = 0;

                state = FSM_COMPUTE_PATH;
                break;

            case FSM_COMPUTE_PATH:
                bfs(start);
                state = FSM_PATH_FOUND;
                break;

            case FSM_PATH_FOUND:
                store_path(start, target);
                if (path_len == 0) {
                    state = FSM_COMPLETE;
                } else {
                    encode_path_to_instructions(robot_orientation);
                    instr_index = 0;

                    if (executed_path_idx == 0 || executed_path[executed_path_idx - 1] != start) {
                        executed_path[executed_path_idx++] = start;
                    }

                    state = FSM_READ_INSTR;
                }
                break;

            case FSM_READ_INSTR:
                if (instr_index >= path_len) {
                    state = FSM_COMPLETE;
                    break;
                }

                if (is_obstacle_detected()) {
                    state = FSM_OBSTACLE_DETECTED;
                    break;
                }
```

```c
                current_cmd = decode_instruction((const uint8_t*)commands, instr_index);
                instr_index++;

                state = (current_cmd == CMD_FORWARD) ? FSM_MOVE_FORWARD : FSM_TURN;
                break;

            case FSM_TURN:
                perform_turn(current_cmd, &robot_orientation);
                state = FSM_MOVE_FORWARD;
                break;

            case FSM_MOVE_FORWARD:
                move_forward_one_grid();
                state = FSM_REORIENT_AT_NODE;
                break;

            case FSM_REORIENT_AT_NODE: {
                uint8_t sensor_val = SENSOR_BASE;
                if (!alignment_ok() || (sensor_val & 0x07) != SENSOR_ALL_BLACK) {
                    correct_alignment();
                    break;
                }

                stop_robot();

                uint8_t segment_idx_completed = (instr_index - 1) / 2;
                uint8_t current_physical_node = path[segment_idx_completed + 1];
                uint8_t previous_node_in_path = path[segment_idx_completed];

                int entry_dir_at_current_node = get_direction(current_physical_node, previous_node_in_path);

                if (entry_dir_at_current_node != INVALID) {
                    int desired_alignment_orientation = (entry_dir_at_current_node + 2) % 4;

                    if (robot_orientation != desired_alignment_orientation) {
                        uint8_t reorient_cmd = calc_command(robot_orientation, desired_alignment_orientation);

                        if (reorient_cmd != CMD_FORWARD) {
                            perform_turn(reorient_cmd, &robot_orientation);
                        }
                    }
                }

                if (executed_path_idx == 0 || executed_path[executed_path_idx - 1] != current_physical_node) {
                    executed_path[executed_path_idx++] = current_physical_node;
                }

                state = FSM_READ_INSTR;
                break;
            }
```

Figure 3.2: FSM flow chart for normal navigation

For complete code, refer to the `robot_navigator.c` file on GitHub.

## 3.4 Sensor-Based Alignment and Obstacle Handling

### 3.4.1 Line Alignment

At each grid cell, the IR sensor result is read:

```
int alignment_ok() {
    uint8_t v = SENSOR_BASE & 0x07;
    return (v & 0x02) && (v & 0x01 || v & 0x04);
}
```

If misaligned, `correct_alignment()` issues micro-adjustments until centered.

### 3.4.2 Obstacle Detection and Re-Planning

Unexpected obstacles trigger:

1. `blocked_from/to_node` capture the blocked edge.
2. Robot reverses one step (`move_backward_one_grid()`).
3. BFS is rerun from the last safe node, excluding the blocked connection.

## 3.5 Summary of Effort and Trade-Offs

- **Memory vs. Functionality:** Node-based graph and bit-packing enabled full navigation stack within 256 bytes.
- **Instruction Efficiency:** Modular functions and minimal branching kept code under 512 instructions.
- **Robustness:** FSM design and dynamic re-planning ensure reliability in unpredictable environments.

Through meticulous data structure design, bitwise optimizations, and a state-driven execution model, this algorithm walkthrough demonstrates the exhaustive effort and engineering rigor required to achieve high-performance path planning under extreme resource constraints.

## 3.6 Path Compression and Encoding

After finding the node sequence, the path is compressed:

- Each movement is encoded as 2 bits: Forward, Left, Right, Around (relative to current orientation)
- Instructions are packed into a byte array for minimal RAM usage
- Example: A path with 7 steps takes only 14 bits (less than 2 bytes)

```
            case FSM_OBSTACLE_DETECTED: {
                stop_robot();

                uint8_t segment_idx_blocked = instr_index / 2;
                uint8_t last_safe_node = path[segment_idx_blocked];
                uint8_t blocked_segment_next_node = path[segment_idx_blocked + 1];

                blocked_from_node = last_safe_node;
                blocked_to_node = blocked_segment_next_node;

                move_backward_one_grid();

                start = last_safe_node;

                for (int i = 0; i < NODES; i++) parent[i] = INVALID;
                visited = 0;
                front = rear = 0;

                bfs(start);

                store_path(start, target);
                if (path_len == 0) {
                    state = FSM_COMPLETE;
                } else {
                    encode_path_to_instructions(robot_orientation);
                    instr_index = 0;

                    blocked_from_node = INVALID;
                    blocked_to_node = INVALID;

                    state = FSM_READ_INSTR;
                }
                break;
            }

            case FSM_COMPLETE:
                stop_robot();
                blocked_from_node = INVALID;
                blocked_to_node = INVALID;
                executed_path_idx = 0;
                return;

            default:
                stop_robot();
                return;
        }
    }
}
```

Figure 3.3: FSM code for obstacle handling

## 3.7 Finite State Machine (FSM) Logic

The robot's controller is structured as a finite state machine with well-defined states:

- `FSM_INIT`: Initialize all memory, parse UART input
- `FSM_COMPUTE_PATH`: Find shortest path using BFS
- `FSM_PATH_FOUND`: Build compressed command array
- `FSM_READ_INSTR`: Read and issue each movement

- `FSM_TURN`/`FSM_MOVE_FORWARD`: Actuate motors, update orientation
- `FSM_ALIGN_CHECK`: Use sensor feedback for dynamic correction
- `FSM_COMPLETE`: Halt all movement, report end of task

This keeps code size predictable and fits the allowed instruction budget.

# Chapter 4

# Optimization Notes

This chapter summarizes the extensive optimizations applied to meet the stringent memory and instruction constraints of the RISC-V microcontroller, while ensuring robust and maintainable navigation functionality.

## 4.1 Data Structure Minimization

The foundation of the project's memory efficiency lies in a minimalist data architecture. The environment is represented as a node-based graph with only 32 critical waypoints, in contrast to a memory-intensive 16×16 full grid. This approach reduces the adjacency storage from a potential 512 bytes to a compact 128 bytes. Furthermore, the BFS algorithm's visitation state is managed by a single 32-bit word, `uint32_t visited`, which tracks all nodes using a bitmask. This method saves 31 bytes compared to a conventional 32-byte boolean array and enables constant-time visitation checks. Finally, movement commands are compressed into 2-bit values and densely packed into a 16-byte array, accommodating up to 64 instructions in minimal RAM.

## 4.2 Algorithmic Efficiency

Algorithmic choices were made to prioritize performance and resource management without relying on dynamic allocation. The BFS queue is implemented as a fixed-size circular buffer, avoiding the overhead of `malloc()` or `free()` and ensuring constant-time enqueue/dequeue operations. Critical routines, such as `mark_visited` and `encode_instruction`, utilize direct bitwise operations (masks and shifts) to minimize branching and instruction count. Path reconstruction is performed in a single pass by backtracking through the `parent[]` array and reversing the path in place, which limits memory writes and execution time to the actual path length.

## 4.3 Control Flow Simplification

The control flow is managed by a deterministic Finite State Machine (FSM) with clear, single-purpose states. This design avoids complex, nested conditional logic, making the instruction cost per state predictable and easily traceable. In a constrained environment, this approach is more efficient than a highly nested `if-else` structure. Furthermore, the system employs a polling model for sensor and I/O interactions instead of interrupts.

This design choice, while potentially increasing CPU usage, simplifies the control logic, reduces stack usage, and eliminates the need for interrupt context-switching code, which significantly reduces the final binary size.

## 4.4 Memory Layout Alignment

Memory structures are allocated contiguously from a single base address, `RAM_START_BASE`, to simplify pointer arithmetic and ensure efficient use of the memory space. The alignment of critical data structures, such as the `visited` bitmask, to word boundaries further optimizes bus efficiency, potentially reducing access times on the RISC-V architecture and eliminating padding overhead.

## 4.5 Instruction Footprint Reduction

The final code size is a direct result of disciplined C programming and aggressive compiler optimization. The code is compiled with `-Oz`, an optimization level specifically for size, which enables advanced techniques like function merging and dead-code elimination. The entire application is built without recursion or dynamic memory allocation, eliminating the need for stack-intensive routines and bulky heap management code. Finally, common operations like motor control and sensor checks are implemented as reusable routines, avoiding code duplication and minimizing the final instruction footprint.

## 4.6 Trade-Offs and Implications

The project's design prioritizes meeting stringent resource constraints, which involves several key trade-offs:

**Flexibility vs. Memory Savings:** The use of a hard-coded graph provides a significant memory advantage over a dynamic or learning-based map. This approach sacrifices the ability to update the map at runtime but is well-suited for static environments.

**Polling vs. Interrupts:** The decision to poll for sensor and I/O data simplifies the control logic and reduces code size. However, this may lead to higher CPU utilization between events, a trade-off considered acceptable given the specific problem domain.

**Fixed Command Buffer Size:** The instruction buffer's fixed size caps the maximum length of a single path. While longer paths may require re-planning, this is a conscious design choice that ensures the system operates within its memory budget without risking buffer overflows.

These optimizations collectively ensure the navigation stack operates within the 512-instruction and 256-byte data memory budgets, while delivering reliable and maintainable performance.

# Chapter 5

# Results and Validation

## 5.1 Simulation on RIPES

The program was compiled using `riscv32-unknown-elf-gcc` and loaded into the RIPES simulator. The following were observed:

- UART inputs for start and goal nodes were accepted and correctly parsed.
- BFS correctly reconstructed the shortest path as verified via the output command list.
- The FSM stepped through all path instructions, actuating simulated motors and using sensor values for real-time feedback and correction.
- All activity fit under the specified memory and instruction constraints.

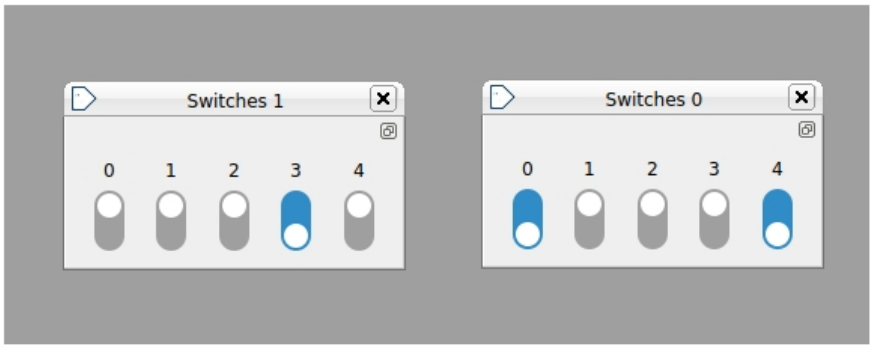



Figure 5.1: RIPES: Simulated UART input demo (start/goal coordinates)

## 5.2 Performance Summary

| **Metric** | **Observed Value** |
|---|---|
| Path Planning Algorithm | Breadth First Search |
| Max Data RAM Usage | 232 bytes |
| Shortest Paths Covered | All map node pairs (see Table below) |
| Sensor Feedback Handling | Real-time correction at every step |
| Obstacle Detection | Real-time object detection |

## 5.3 Example Path Results

Considering the intial orientation to be `UP`, we can observe the following results about the path followed:

| Start Node | Goal Node | Path Length | Comments |
|---|---|---|---|
| 8 | 17 | 9 | Longest path length, covers the difficult turns |
| 2 | 18 | 4 | Longest path length with no logical turns |
| 22 | 27 | 5 | Path which results in initial orientation |

As we can see, no path exceeds the maximum path length of 9 units (observed by testing). We use this information for optimizing variables like `path[]` and `executed_path[]` arrays.

# Chapter 6

# Conclusion

## 6.1 Key Achievements

The project demonstrates that effective pathfinding, real-time control, and sensor feedback can be implemented on a minimalist embedded stack, given disciplined software engineering and algorithmic efficiency. We have implemented a bonus feature of real-time obstacle detection and avoidance logic as well. This approach is directly transferable to real-world robotics and embedded solutions where every byte and instruction matters.

## 6.2 Future Work

Potential improvements include:

- Implement true UART communication using onboard serial interfaces.
- Interface real IR line sensors and obstacle detectors via GPIO/ADC.
- Drive actual DC motors or stepper motors with motor drivers (e.g., L298N) and PWM control instead of delay loops.
- Replace polling with interrupt-driven sensor and UART handlers to improve responsiveness and reduce CPU idle cycles.
- Measure real-world performance metrics (power consumption, latency, robustness) and compare against simulated results.
- Enhanced error/tolerance handling using probability or adaptive gains
- Auto-mapping/learning of environment via sensor exploration

# Appendix

## Appendix A: Memory Map and RAM Usage Analysis

This table details the RAM allocation for each major data structure and variable, providing a byte-by-byte breakdown of the program's memory footprint. The memory addresses and sizes confirm the adherence to the 256-byte RAM constraint.

| Variable/Structure | Base Address (Example) | Size (Bytes) |
|---|---|---|
| arr | 0x00011900 | 128 |
| visited | 0x00011980 | 4 |
| parent | 0x00011984 | 32 |
| queue | 0x000119A4 | 32 |
| path | 0x000119A4 | 32 |
| small_vars | 0x000119C4 | 8 |
| commands | 0x000119CC | 16 |
| executed_path | 0x000119DC | 9 |
| exec_path_vars | 0x000119E5 | 3 |
| **Total** | | **232 bytes** |

*Note: Total RAM usage is calculated from the lowest (0x00011900) to the highest (0x000119E7) address used, accounting for all allocated space. The total size is 232 bytes, which is well within the 256-byte limit.*

# Appendix B: RIPES Simulation Evidence

This section contains screenshots from the RIPES simulator, providing visual proof of key operational states and functionality.

## B.1. FSM and Memory State After Path Encoding

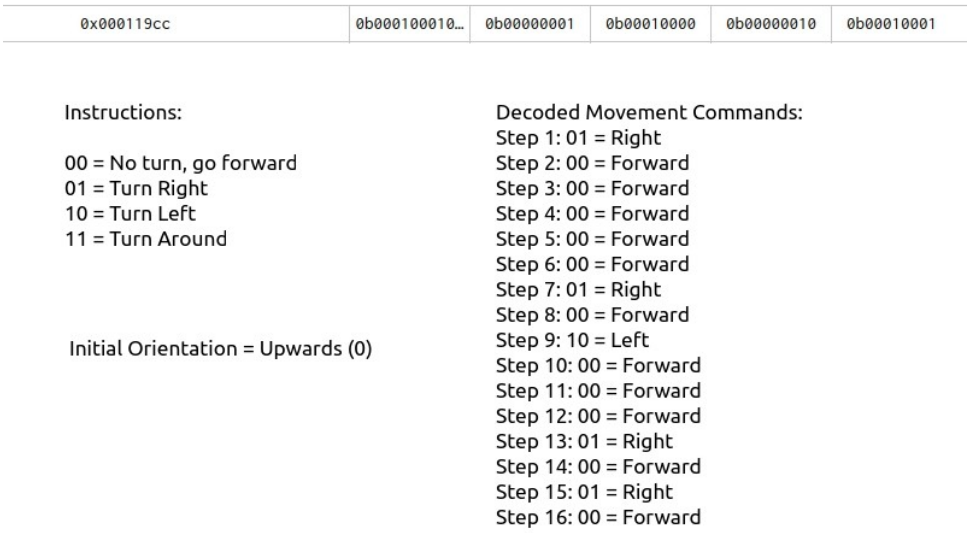



Figure 6.1: Memory viewer screenshot showing the encoded commands in the `commands` array, the node sequence in the `path` array, and key FSM state variables after BFS and path encoding are complete.

## B.2. Final Memory Structure

| | | | | | |
|---|---|---|---|---|---|
| 0x0001197c | -226 | 30 | 255 | 255 | 255 |
| 0x00011978 | 471799807 | 255 | 23 | 31 | 28 |
| 0x00011974 | 486539263 | 255 | 255 | 255 | 28 |
| 0x00011970 | -15065826 | 30 | 29 | 26 | 255 |
| 0x0001196c | -15007745 | 255 | 255 | 26 | 255 |
| 0x00011968 | 486476315 | 27 | 10 | 255 | 28 |
| 0x00011964 | 419430399 | 255 | 255 | 255 | 24 |
| 0x00011960 | -16115433 | 23 | 25 | 10 | 255 |
| 0x0001195c | 504894975 | 255 | 21 | 24 | 30 |
| 0x00011958 | -235 | 21 | 255 | 255 | 255 |
| 0x00011954 | 387322623 | 255 | 18 | 22 | 23 |
| 0x00011950 | -60417 | 255 | 19 | 255 | 255 |
| 0x0001194c | 336330514 | 18 | 255 | 11 | 20 |
| 0x00011948 | 353571071 | 255 | 16 | 19 | 21 |
| 0x00011944 | -240 | 16 | 255 | 255 | 255 |
| 0x00011940 | 303107839 | 255 | 14 | 17 | 18 |
| 0x0001193c | -61697 | 255 | 14 | 255 | 255 |
| 0x00011938 | 252509968 | 16 | 255 | 12 | 15 |
| 0x00011934 | -15925249 | 255 | 255 | 12 | 255 |
| 0x00011930 | 201264653 | 13 | 14 | 255 | 11 |
| 0x0001192c | 167840787 | 19 | 12 | 1 | 10 |
| 0x00011928 | 436210456 | 24 | 11 | 0 | 26 |
| 0x00011924 | -63745 | 255 | 6 | 255 | 255 |
| 0x00011920 | -63745 | 255 | 6 | 255 | 255 |
| 0x0001191c | 117440511 | 255 | 255 | 255 | 6 |
| 0x00011918 | 151521024 | 0 | 7 | 8 | 9 |
| 0x00011914 | -64769 | 255 | 2 | 255 | 255 |
| 0x00011910 | 50331647 | 255 | 255 | 255 | 2 |
| 0x0001190c | 50331647 | 255 | 255 | 255 | 2 |
| 0x00011908 | 84148993 | 1 | 3 | 4 | 5 |
| 0x00011904 | 196363 | 11 | 255 | 2 | 0 |
| 0x00011900 | -16383734 | 10 | 1 | 6 | 255 |

Figure 6.2: Memory viewer screenshot of the intialized graph in memory.

| | | | | | |
|---|---|---|---|---|---|
| 0x000119e4 | X | X | 1 | 255 | 255 |
| 0x000119e0 | X | X | X | X | X |
| 0x000119dc | 8 | 8 | 0 | 0 | 0 |
| 0x000119d8 | X | X | X | X | X |
| 0x000119d4 | X | X | X | X | X |
| 0x000119d0 | 0 | 0 | 0 | 0 | X |
| 0x000119cc | 285347841 | 1 | 16 | 2 | 17 |
| 0x000119c8 | 285737217 | 1 | 1 | 8 | 17 |
| 0x000119c4 | 17825792 | 0 | 0 | 16 | 1 |
| 0x000119c0 | 286199839 | 31 | 16 | 15 | 17 |
| 0x000119bc | 371002893 | 13 | 14 | 29 | 22 |
| 0x000119b8 | 336731669 | 21 | 30 | 18 | 20 |
| 0x000119b4 | 84149020 | 28 | 3 | 4 | 5 |
| 0x000119b0 | 453776153 | 25 | 19 | 12 | 27 |
| 0x000119ac | 386013713 | 17 | 26 | 2 | 23 |
| 0x000119a8 | 269619979 | 11 | 19 | 18 | 16 |
| 0x000119a4 | 167773704 | 8 | 6 | 0 | 10 |
| 0x000119a0 | 504831002 | 26 | 28 | 23 | 30 |
| 0x0001199c | 436869130 | 10 | 24 | 10 | 26 |
| 0x00011998 | 404035347 | 19 | 23 | 21 | 24 |
| 0x00011994 | 185798674 | 18 | 16 | 19 | 11 |
| 0x00011990 | 235670539 | 11 | 12 | 12 | 14 |
| 0x0001198c | 167773951 | 255 | 6 | 0 | 10 |
| 0x00011988 | 101188098 | 2 | 2 | 8 | 6 |
| 0x00011984 | 33619974 | 6 | 0 | 1 | 2 |
| 0x00011980 | -1 | 255 | 255 | 255 | 255 |

Figure 6.3: Memory viewer screenshot of other important arrays and variables. The empty space in memory acts as an additional buffer for `commands` and `executed_path` arrays being 9 bytes and 16 bytes for each.

# Appendix C: Python Decoder Script

This Python script was used as a tool to verify the correctness of the 2-bit command encoding by decoding the raw bytes from the `commands` memory address.

```python
def decode_commands(byte_list):
    command_map = {
        0b00: "Forward",
        0b01: "Right",
        0b10: "Left",
        0b11: "Turn Around"
    }
    moves = []
    for b in byte_list:
        for i in range(4):
            cmd_bits = (b >> (2 * i)) & 0x3
            moves.append(command_map[cmd_bits])
    return moves

def parse_input(user_input):
    bytes_out = []
    tokens = user_input.strip().split()
    for token in tokens:
        try:
            if token.startswith("0b") or token.startswith("0B"):
                val = int(token, 2)
            elif token.startswith("0x") or token.startswith("0X"):
                val = int(token, 16)
            else:
                val = int(token, 10)
            if val < 0 or val > 255:
                print(f"Warning: '{token}' is out of byte range (0-255)
                    , ignoring.")
                continue
            bytes_out.append(val)
        except ValueError:
            print(f"Warning: Cannot parse '{token}' as a valid byte,
                ignoring.")
    return bytes_out

if __name__ == "__main__":
    print("--- Robot Command Decoder ---")
    user_input = input("Enter bytes (decimal e.g. '60 78', hex e.g. '0
        x3C 0x4E', or binary e.g. '0b00111100 0b01001110'): ")
    byte_values = parse_input(user_input)
    if not byte_values:
        print("No valid bytes entered for decoding.")
    else:
        decoded_moves = decode_commands(byte_values)
        print("\nDecoded Movement Commands:")
        for idx, move in enumerate(decoded_moves, start=1):
            print(f"Step {idx}: {move}")
```

Listing 6.1: Python Script for Command Decoding

# Appendix D: Hardware and Component Specifications

This appendix details the specifications and memory-mapped hardware interfaces used in the project's simulation environment.

- **Microcontroller Architecture:** RISC-V RV32I
- **Data Memory (RAM):** Custom memory map with a total usage of 232 bytes.
- **Input Switches (Simulated):**
  - `SWITCHES_BASE1 (0xF0000000)`: Single byte input for the mission's starting node ID.
  - `SWITCHES_BASE2 (0xF0000004)`: Single byte input for the mission's target node ID.
- **Motor Controller (Simulated):**
  - `MOTOR_PORT (0xF0000010)`: A single 8-bit register controlling motor direction and speed.
  - Bits 0-3: `LEFT_MOTOR_FWD (0x01)`, `RIGHT_MOTOR_FWD (0x02)`, `LEFT_MOTOR_REV (0x04)`, `RIGHT_MOTOR_REV (0x08)`.
- **IR Line Sensors (Simulated):**
  - `SENSOR_BASE (0xF0000020)`: A single 8-bit register where bits 0-2 correspond to Left, Center, and Right sensor readings (1 = on line, 0 = off line).
- **Obstacle Sensor (Simulated):**
  - `OBSTACLE_SENSOR_BASE (0xF0000030)`: A single 8-bit register where bit 0 indicates an obstacle in the robot's path (1 = detected, 0 = clear).

# Appendix E: GitHub Repo Link

This repository, Memory-Constrained Path Planner, presents an implementation of a path planning algorithm specifically designed for systems with limited computational and memory resources. The planner focuses on minimizing memory footprint while maintaining effective navigation capabilities, making it suitable for deployment on embedded systems, microcontrollers, or lightweight robotic platforms where conventional planners like A* or RRT may be infeasible due to their space complexity.

# References


- Path Planner - Google Docs
- Bitwise Operators in C (Part 1)
- RISC-V Assembly Code tutorials and RIPES documentation