E-puck Motion Controller Technical Documentation

Executive Summary

This document provides comprehensive technical documentation of the `motion_control.h` implementation for controlling an E-puck robot in a Webots maze environment. The controller enables autonomous navigation through a 0.25m tile-based maze using precise forward movement, accurate 90° turns, wall-following for corridor centering, and robust heading estimation through odometry.

---

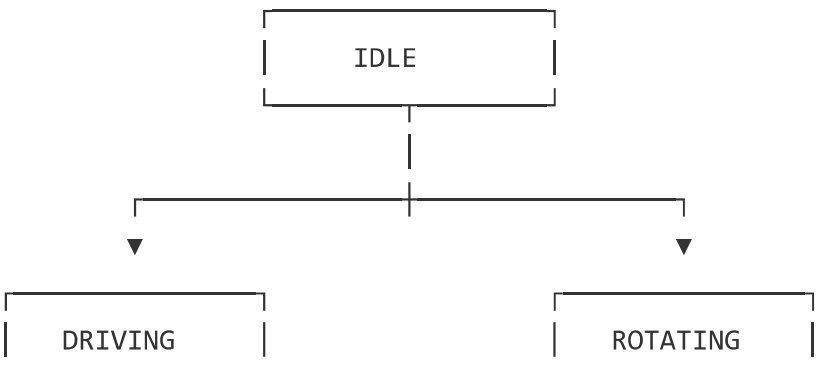
1. System Architecture Overview

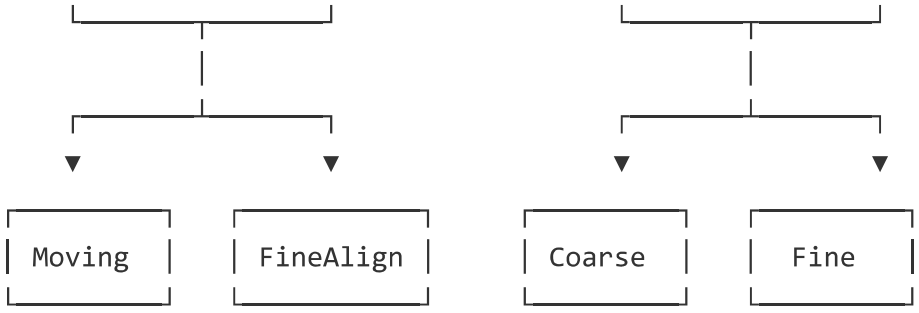
The motion controller follows a hierarchical state machine architecture with multiple control loops operating at different frequencies and precision levels.

1.1 Core Components

Component	Purpose
`MazeConfig`	Physical constants and calibration parameters
`PIDParams`	Tuning coefficients for all PID controllers
`PID` class	Reusable PID controller with advanced features
`MovingController`	Main state machine coordinating all behaviors

1.2 State Machine Hierarchy





2. Physical Constants and Calibration (MazeConfig)

2.1 Robot Physical Parameters

```
TILE_SIZE = 0.25          // Maze tile dimension (meters)
ROBOT_WIDTH = 0.074       // E-puck chassis width (meters)
ROBOT_RADIUS = 0.037      // Half of chassis width (meters)
WHEEL_RADIUS = 0.02001    // Wheel radius for odometry (meters)
AXLE_LENGTH = 0.052       // Distance between wheel centers (meters)
MAX_SPEED = 6.28          // Maximum motor velocity (rad/s)
```

2.2 Navigation Parameters

```
WALL_CLEARANCE = 0.088    // Desired distance from wall (meters)
TURN_90_RAD = 1.5708      // 90° in radians (π/2)
INITIAL_HEADING = 1.5709  // North-facing start orientation
```

2.3 Calibration Constants (Critical for Accuracy)

DRIFT_PER_METER = 0.00406

This constant compensates for systematic heading drift during forward motion. The E-puck experiences clockwise drift due to mechanical asymmetries. The value represents radians of counterclockwise correction applied per meter traveled.

```
correctedHeading = rawHeading - (distanceTraveled × DRIFT_PER_METER)
```

TURN_SLIP_FACTOR = 0.913

During in-place rotation, wheel slip causes the encoders to report more rotation than actually occurs. This factor scales down the odometry-derived angular change.

```
Measured error = 0.137 rad per 90° turn
Slip factor = 1 - (0.137 / 1.5708) ≈ 0.913
```

---

3. PID Controller Implementation

3.1 Controller Structure

The `PID` class implements a professional-grade discrete PID controller with:

1. **Derivative Filtering** - Low-pass filter on derivative term to reduce noise amplification
2. **Anti-Windup** - Integral clamping to prevent accumulation during saturation
3. **Output Rate Limiting** - Prevents sudden output changes that could cause mechanical stress
4. **Output Saturation** - Enforces maximum motor speed limits

3.2 Mathematical Formulation

Standard PID Equation:

```
u(t) = Kp·e(t) + Ki·∫e(τ)dτ + Kd·(de/dt)
```

Discrete Implementation with Filtering:

```
// Proportional term
pOut = kp_ × error

// Integral term with anti-windup
integral_ += error × dt
integral_ = clamp(integral_, -maxIntegral_, +maxIntegral_)
iOut = ki_ × integral_

// Derivative term with exponential moving average filter
rawDerivative = (error - prevError_) / dt
filteredDerivative = α × rawDerivative + (1-α) × prevDerivative_
dOut = kd_ × filteredDerivative
```

Where `α = filterAlpha_ = 0.2` determines the filter cutoff frequency.

3.3 Rate Limiting

```
maxChange = maxRamp_ × dt  // maxRamp_ = 20.0
output = clamp(output, prevOutput_ - maxChange, prevOutput_ + maxChange)
```

---

4. Heading Estimation System

4.1 Pure Odometry Approach

The controller uses pure wheel odometry for heading estimation (gyro fusion is disabled as it caused instability in testing).

Differential Drive Kinematics:

```
Δθ = (ΔR - ΔL) / L

Where:
- ΔR = Right wheel arc length = (encoder_change) × WHEEL_RADIUS
- ΔL = Left wheel arc length
- L = AXLE_LENGTH = 0.052m
```

4.2 State-Dependent Processing

During Forward Motion:

```
deltaOdom = (dR - dL) / AXLE_LENGTH
distStep = (dL + dR) / 2.0
driftCorrection = distStep × DRIFT_PER_METER
odomHeading_ = normalize(odomHeading_ + deltaOdom - driftCorrection)
```

During Rotation:

```
deltaOdom = (dR - dL) / AXLE_LENGTH
deltaOdom *= TURN_SLIP_FACTOR  // Apply slip compensation
odomHeading_ = normalize(odomHeading_ + deltaOdom)
```

---

5. Wall Centering System

5.1 Sensor Configuration

The E-puck's proximity sensors are indexed 0-7 around the robot. For wall detection:

- **Sensor 5** (ps5): Left side
- **Sensor 2** (ps2): Right side
- **Sensors 0, 7**: Front (for obstacle detection)

5.2 Three-Mode Wall Correction

Mode 1: Corridor (Both walls present)

```
if (dL < 0.15 && dR < 0.15) {
    error = (dL - dR) / 2.0
    correction = error × WALL_GAIN
}
```

This centers the robot between walls. If left wall is closer (dL < dR), correction is negative (steer right).

Mode 2: Left wall only

```
if (dL < WALL_CLEARANCE) {
    correction = -(WALL_CLEARANCE - dL) × WALL_GAIN
}
```

Mode 3: Right wall only

```
if (dR < WALL_CLEARANCE) {
    correction = +(WALL_CLEARANCE - dR) × WALL_GAIN
}
```

5.3 Transition Smoothing (Accepted Change)

Problem Addressed: When a wall suddenly appears or disappears, the correction value can jump from near-zero to a large value, causing visible jerk.

Solution Implemented:

```
// Only rate-limit when a large jump occurs (> 0.05)
double change = rawCorrection - prevWallCorrection_
if (abs(change) > 0.05) {
    constexpr double maxChange = 0.03
    if (change > maxChange)
        rawCorrection = prevWallCorrection_ + maxChange
    else if (change < -maxChange)
        rawCorrection = prevWallCorrection_ - maxChange
```

```
}
prevWallCorrection_ = rawCorrection
```

6. Speed Control System

6.1 Adaptive Base Speed

```
if (inCorridor() && frontClearance() > 0.30 && remaining > 0.10) {
    driveMode_ = FastCorridor
    speed = MAX_SPEED  // 6.28 rad/s
} else {
    speed = 6.0  // Near maximum speed
}
```

6.2 Progressive Deceleration Zones

Remaining Distance	Maximum Speed
> 0.15m	Full speed
0.08m - 0.15m	4.5 rad/s
0.03m - 0.08m	2.5 rad/s
< 0.03m	1.2 rad/s

7. Driving State Machine

7.1 Phase 1: Moving

```
// Distance Control
distOutput = distPID_.compute(remaining, dt)
speedCmd = min(baseSpeed, abs(distOutput))

// Heading Maintenance
headingError = normalize(targetHeading_ - fusedHeading_)
turnCorrection = anglePID_.compute(headingError, dt)
```

```
// Wall Following + Motor Command
wallCorrection = getWallCorrection()
totalTurn = turnCorrection + wallCorrection
leftSpeed = speedCmd - totalTurn
rightSpeed = speedCmd + totalTurn
```

7.2 Phase 2: Fine Alignment

When `remaining < 0.005m`, switches to FineAlign phase:

```
turnSpeed = fineRotPID_.compute(headingError, dt)
turnSpeed = clamp(turnSpeed, ±0.5)  // Slow, precise
setMotors(-turnSpeed, turnSpeed)     // In-place rotation
```

Completion: Error < 0.001 rad for 3 consecutive cycles

---

8. Rotation State Machine

8.1 Two-Phase Turning Strategy

A single-phase PID cannot achieve both fast turns and precise alignment. The two-phase approach uses different controllers optimized for each objective.

8.2 Phase 1: Coarse Turn

- **Controller:** rotPID_ with KP=1.5, KI=0.0, KD=0.4
- **Speed Limit:** ±3.0 rad/s (fast rotation)
- **Exit Condition:** |error| < 0.03 rad (~1.7°)

8.3 Phase 2: Fine Alignment

- **Controller:** fineRotPID_ with KP=3.0, KI=0.1, KD=0.8
- **Speed Limit:** ±0.5 rad/s (slow, precise)
- **Completion:** Error < 0.001 rad for 5 consecutive cycles

8.4 Settle Counter Mechanism

```
if (abs(headingError) < FINE_TOLERANCE) {
    settleCounter_++
    if (settleCounter_ >= SETTLE_CYCLES)
        // Turn complete
```

```
} else {
    settleCounter_ = 0  // Reset if error exceeds tolerance
}
```

---

9. Debug Logging System

The controller maintains comprehensive debug logs in `debug_report.txt`:

9.1 Driving Log Entry

```
--- TIMESTAMP: XX.XXXX ---
State: DRIVING | Mode: NORM/FAST
Dist: X.XXXXX | Rem: X.XXXXX | Target: X.XXXXX
Yaw: X.XXXX | Tgt: X.XXXX | Err: X.XXXX
TurnCorr: X.XXXX | WallCorr: X.XXXX
Speed: X.XXXX | Motors(L/R): X.XXXX / X.XXXX
```

9.2 Rotation Log Entry

```
--- TIMESTAMP: XX.XXXX ---
State: ROTATING | Phase: COARSE/FINE
Yaw: X.XXXXX | Tgt: X.XXXXX | Err: X.XXXXX
TurnSpeed: X.XXXXX | SettleCount: X / 5
```

---

10. Accepted Code Changes

10.1 Wall Correction Transition Smoothing

Files Modified: motion_control.h

Components Added:

- `prevWallCorrection_` member variable
- Rate-limiting logic in `getWallCorrection()`

Problem Solved: Wall correction jumping 38× (0.005 → 0.192) at corridor intersections caused visible jerk.

Solution: Rate-limit changes to max 0.03 per timestep when jump > 0.05. Normal corrections unaffected.

Impact:

- **Speed:** Unaffected
- **Accuracy:** Unaffected
- **Smoothness:** Significantly improved

---

11. Known Limitations

11.1 Accumulated Heading Error

After 19 movements, ~0.01 rad (0.57°) error accumulates due to:

- Each movement completing with up to 0.001 rad residual error
- Errors sum rather than canceling (systematic bias)

Potential Improvements: Tune DRIFT_PER_METER and TURN_SLIP_FACTOR more precisely.

11.2 Encoder Quantization

At fine tolerances (<0.001 rad), encoder discretization can cause oscillation between adjacent quantization levels. This occasionally delays settling but does not prevent completion.

---

12. API Reference

Method	Description
`moveForward(tiles)`	Move forward by specified number of tiles (0.25m each)
`moveForwardMeters(meters)`	Move forward by exact distance in meters
`turnLeft()`	Execute 90° counterclockwise turn
`turnRight()`	Execute 90° clockwise turn
`rotate(radians)`	Execute turn by specified angle
`stop()`	Immediately halt all motors
`isBusy()`	Returns true if movement in progress

`getState()`	Returns current state (Idle/Driving/Rotating)
`update(dt)`	Main control loop - call once per timestep

12.1 Usage Example

```
Sensing* sensing = new Sensing(robot, timeStep);
MovingController* motion = new MovingController(robot, sensing);

// Main control loop
while (robot->step(timeStep) != -1) {
    sensing->update();
    motion->update(timeStep / 1000.0);

    if (!motion->isBusy()) {
        motion->moveForward(3);  // Move 3 tiles
    }
}
```

Document generated: December 8, 2025
Controller Version: motion_control.h (581 lines)