Hardware Integration Guide - Material Handling System

1. System Requirements

1.1 Hardware Platform Requirements

Robot Platform

- **Differential Drive Robot** (preferred) or **Mecanum Drive**
- **Minimum Payload**: 5-10 kg for material handling
- **Operating Speed**: 0.1-1.0 m/s
- **Turning Radius**: < 0.5m for warehouse navigation
- **Battery Life**: 4+ hours continuous operation

Recommended Platforms

- **TurtleBot3 Burger/Waffle**: Educational/research
- **Clearpath Jackal**: Industrial applications
- **Custom AGV**: Heavy-duty material handling
- **ROSbot 2.0**: Mid-range applications

Onboard Computer

- **Minimum**: Raspberry Pi 4 (4GB RAM)
- **Recommended**: Intel NUC or NVIDIA Jetson Nano
- **Industrial**: Advantech or Kontron industrial PCs
- **Requirements**: Ubuntu 20.04/22.04, ROS2 Humble/Iron

1.2 Sensor Requirements

Essential Sensors

- **Wheel Encoders**: High-resolution (1000+ PPR)
- **IMU**: 9-DOF for orientation feedback
- **Emergency Stop**: Physical button for safety

Navigation Sensors (Optional but Recommended)

- **LiDAR**: 2D scanning for obstacle detection
- **RGB-D Camera**: Intel RealSense or similar
- **Ultrasonic Sensors**: Close-range obstacle detection

Material Handling Sensors

- **Weight Sensors**: Load cell for pickup confirmation
- **Proximity Sensors**: Item detection at pickup/dropoff
- **Camera**: QR code/barcode scanning for inventory

2. Software Stack Setup

2.1 ROS2 Installation

Base System Setup

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install ROS2 Humble
sudo apt install software-properties-common
sudo add-apt-repository universe
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key | sudo apt-key add -
sudo sh -c 'echo "deb [arch=amd64] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" > /etc/apt/sources.list.d/ros2-latest.list'
sudo apt update
sudo apt install ros-humble-desktop

# Source ROS2
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Required ROS2 Packages

```
# Essential packages
sudo apt install ros-humble-robot-localization \
	ros-humble-navigation2 \
	ros-humble-nav2-bringup \
	ros-humble-slam-toolbox \
	ros-humble-robot-state-publisher \
	ros-humble-joint-state-publisher \
	ros-humble-tf2-ros \
	ros-humble-tf2-geometry-msgs

# Web interface packages
sudo apt install ros-humble-rosbridge-suite \
	ros-humble-web-video-server

# Hardware interface packages
sudo apt install ros-humble-hardware-interface \
	ros-humble-controller-manager \
	ros-humble-diff-drive-controller \
	ros-humble-joint-state-broadcaster
```

2.2 Robot Description Setup

Create Robot URDF

```
<!-- material_handling_robot.urdf -->
<robot name="material_handling_robot">
 <link name="base_link">
  <visual>
   <geometry>
    <box size="0.6 0.4 0.2"/>
   </geometry>
   <material name="blue">
    <color rgba="0 0 1 1"/>
   </material>
  </visual>
  <collision>
   <geometry>
    <box size="0.6 0.4 0.2"/>
   </geometry>
  </collision>
  <inertial>
   <mass value="20"/>
   <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
  </inertial>
 </link>

 <!-- Add wheels, sensors, etc. -->
</robot>
```

Robot Configuration Package

```
# Create robot package
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake material_handling_robot
```

3. Hardware Interface Implementation

3.1 Motor Controller Interface

Arduino/Microcontroller Code

```
// motor_controller.ino
#include <ros.h>
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include <Encoder.h>

// Motor pins
const int LEFT_MOTOR_PWM = 3;
const int LEFT_MOTOR_DIR = 4;
const int RIGHT_MOTOR_PWM = 5;
const int RIGHT_MOTOR_DIR = 6;

// Encoder pins
Encoder leftEncoder(18, 19);
Encoder rightEncoder(20, 21);

// Robot parameters
const float WHEEL_RADIUS = 0.08;  // meters
const float WHEEL_BASE = 0.32;    // meters
const int ENCODER_CPR = 1000;     // counts per revolution

ros::NodeHandle nh;
nav_msgs::Odometry odom_msg;
ros::Publisher odom_pub("odom", &odom_msg);

void cmd_vel_callback(const geometry_msgs::Twist& msg) {
  float linear = msg.linear.x;
  float angular = msg.angular.z;

  // Calculate wheel velocities
  float left_vel = linear - (angular * WHEEL_BASE / 2.0);
  float right_vel = linear + (angular * WHEEL_BASE / 2.0);

  // Convert to PWM values
  int left_pwm = constrain(left_vel * 255 / 1.0, -255, 255);
  int right_pwm = constrain(right_vel * 255 / 1.0, -255, 255);

  // Set motor directions and speeds
  digitalWrite(LEFT_MOTOR_DIR, left_pwm >= 0 ? HIGH : LOW);
```

```
  digitalWrite(RIGHT_MOTOR_DIR, right_pwm >= 0 ? HIGH : LOW);
  analogWrite(LEFT_MOTOR_PWM, abs(left_pwm));
  analogWrite(RIGHT_MOTOR_PWM, abs(right_pwm));
}

ros::Subscriber<geometry_msgs::Twist> cmd_vel_sub("cmd_vel", cmd_vel_callback);

void setup() {
  nh.initNode();
  nh.subscribe(cmd_vel_sub);
  nh.advertise(odom_pub);

  // Initialize motor pins
  pinMode(LEFT_MOTOR_PWM, OUTPUT);
  pinMode(LEFT_MOTOR_DIR, OUTPUT);
  pinMode(RIGHT_MOTOR_PWM, OUTPUT);
  pinMode(RIGHT_MOTOR_DIR, OUTPUT);
}

void loop() {
  // Read encoders and publish odometry
  long left_count = leftEncoder.read();
  long right_count = rightEncoder.read();

  // Calculate odometry (implement odometry calculation)
  // ... odometry calculation code ...

  odom_pub.publish(&odom_msg);
  nh.spinOnce();
  delay(50);  // 20Hz update rate
}
```

3.2 ROS2 Hardware Interface

Hardware Interface Node

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from sensor_msgs.msg import JointState
import serial
import json
```

```python
class MaterialHandlingHardwareInterface(Node):
    def __init__(self):
        super().__init__('material_handling_hardware')

        # Serial connection to microcontroller
        self.serial_port = serial.Serial('/dev/ttyUSB0', 115200)

        # Publishers
        self.odom_pub = self.create_publisher(Odometry, 'odom', 10)
        self.joint_state_pub = self.create_publisher(JointState, 'joint_states', 10)

        # Subscribers
        self.cmd_vel_sub = self.create_subscription(
            Twist, 'cmd_vel', self.cmd_vel_callback, 10)

        # Timers
        self.timer = self.create_timer(0.05, self.hardware_update)  # 20Hz

        self.get_logger().info('Hardware interface initialized')

    def cmd_vel_callback(self, msg):
        # Send velocity command to microcontroller
        cmd = {
            'linear': msg.linear.x,
            'angular': msg.angular.z
        }
        self.serial_port.write(json.dumps(cmd).encode() + b'\n')

    def hardware_update(self):
        # Read sensor data from microcontroller
        if self.serial_port.in_waiting > 0:
            try:
                data = self.serial_port.readline().decode().strip()
                sensor_data = json.loads(data)

                # Publish odometry
                self.publish_odometry(sensor_data)

            except Exception as e:
                self.get_logger().error(f'Serial communication error: {e}')

    def publish_odometry(self, data):
        # Implement odometry message creation and publishing
        odom_msg = Odometry()
        odom_msg.header.stamp = self.get_clock().now().to_msg()
        odom_msg.header.frame_id = 'odom'
        odom_msg.child_frame_id = 'base_link'
```

```
        # Fill in position and velocity from sensor data
        # ... odometry message population ...

        self.odom_pub.publish(odom_msg)

def main(args=None):
    rclpy.init(args=args)
    node = MaterialHandlingHardwareInterface()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

4. Network Configuration

4.1 ROSBridge Setup

Launch ROSBridge Server

```
# Create launch file
mkdir -p ~/ros2_ws/src/material_handling_robot/launch
<!-- rosbridge.launch.py -->
<launch>
  <node pkg="rosbridge_server" exec="rosbridge_websocket" name="rosbridge_websocket">
    <param name="port" value="9090"/>
    <param name="address" value="0.0.0.0"/>
  </node>

  <node pkg="web_video_server" exec="web_video_server" name="web_video_server">
    <param name="port" value="8080"/>
    <param name="address" value="0.0.0.0"/>
  </node>
</launch>
```

4.2 Network Security

Firewall Configuration

```
# Allow ROSBridge and web server ports
sudo ufw allow 9090/tcp
```

```
sudo ufw allow 8080/tcp
sudo ufw allow 22/tcp  # SSH access

# Enable firewall
sudo ufw enable
```

Access Control

```
# Create user for web interface
sudo adduser robot_operator
sudo usermod -aG dialout robot_operator  # Serial port access
```

5. Calibration Procedures

5.1 Odometry Calibration

Wheel Diameter Calibration

```
#!/usr/bin/env python3
# calibrate_wheels.py
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import time

class WheelCalibration(Node):
  def __init__(self):
    super().__init__('wheel_calibration')
    self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)
    self.odom_sub  =  self.create_subscription(Odometry,  'odom',  self.odom_callback,
10)
    self.start_position = None
    self.current_position = None

  def calibrate_linear(self, distance=1.0, speed=0.2):
    """Calibrate linear movement"""
    self.get_logger().info(f'Calibrating linear movement: {distance}m at {speed}m/s')

    # Record start position
    self.start_position = self.current_position

    # Move forward
```

```
    twist = Twist()
    twist.linear.x = speed

    start_time = time.time()
    while (time.time() - start_time) < (distance / speed):
        self.cmd_vel_pub.publish(twist)
        rclpy.spin_once(self, timeout_sec=0.1)

    # Stop
    twist.linear.x = 0.0
    self.cmd_vel_pub.publish(twist)

    # Calculate actual distance
    if self.start_position and self.current_position:
        actual_distance = ((self.current_position.x - self.start_position.x)**2 +
                           (self.current_position.y - self.start_position.y)**2)**0.5

        scale_factor = distance / actual_distance
        self.get_logger().info(f'Calibration factor: {scale_factor:.4f}')
        return scale_factor

    return 1.0

def odom_callback(self, msg):
    self.current_position = msg.pose.pose.position
```

5.2 Material Handling Calibration

Pickup/Dropoff Position Calibration

```
# calibrate_positions.py
def calibrate_pickup_position(location_name):
    """Manually drive robot to pickup position and record coordinates"""
    input(f"Drive robot to {location_name} pickup position and press Enter...")

    # Record current position
    current_pos = get_robot_position()

    # Update location database
    locations[location_name] = {
        'x': current_pos.x,
        'y': current_pos.y,
        'name': location_name
    }
```

```
    print(f"Calibrated {location_name}: ({current_pos.x:.3f}, {current_pos.y:.3f})")

    # Test approach
    test_approach_to_location(location_name)
```

---

6. Safety Implementation

6.1 Emergency Stop System

Hardware E-Stop

```
#!/usr/bin/env python3
import RPi.GPIO as GPIO
from geometry_msgs.msg import Twist

class EmergencyStop:
    def __init__(self):
        self.estop_pin = 18
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.estop_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
        GPIO.add_event_detect(self.estop_pin,                GPIO.FALLING,
callback=self.estop_callback)

        self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)
        self.estop_active = False

    def estop_callback(self, channel):
        self.estop_active = True
        self.get_logger().error('EMERGENCY STOP ACTIVATED')

        # Stop robot immediately
        stop_cmd = Twist()
        self.cmd_vel_pub.publish(stop_cmd)

        # Disable motor power (if possible)
        self.disable_motors()

    def disable_motors(self):
        # Send disable command to motor controller
        pass
```

6.2 Obstacle Detection

LiDAR Integration

```
#!/usr/bin/env python3
import rclpy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

class ObstacleDetector(Node):
    def __init__(self):
        super().__init__('obstacle_detector')

        self.laser_sub = self.create_subscription(LaserScan, 'scan', self.laser_callback, 10)
        self.cmd_vel_sub = self.create_subscription(Twist, 'cmd_vel_raw', self.cmd_vel_callback, 10)
        self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', 10)

        self.safety_distance = 0.5  # meters
        self.obstacle_detected = False

    def laser_callback(self, msg):
        # Check for obstacles in front of robot
        front_ranges = msg.ranges[len(msg.ranges)//4:3*len(msg.ranges)//4]
        min_distance = min(front_ranges)

        self.obstacle_detected = min_distance < self.safety_distance

        if self.obstacle_detected:
            self.get_logger().warn(f'Obstacle detected at {min_distance:.2f}m')

    def cmd_vel_callback(self, msg):
        # Filter velocity commands based on obstacle detection
        if self.obstacle_detected and msg.linear.x > 0:
            # Stop forward movement if obstacle detected
            filtered_cmd = Twist()
            filtered_cmd.angular.z = msg.angular.z  # Allow turning
            self.cmd_vel_pub.publish(filtered_cmd)
        else:
            self.cmd_vel_pub.publish(msg)
```

7. Deployment Checklist

7.1 Pre-Deployment Testing

Hardware Verification

- [ ] Motor controllers respond to commands
- [ ] Encoders provide accurate feedback
- [ ] Emergency stop functions correctly
- [ ] Battery provides adequate runtime
- [ ] All sensors initialized and publishing data

Software Verification

- [ ] ROS2 nodes start correctly
- [ ] ROSBridge server accessible from web interface
- [ ] Odometry accuracy verified
- [ ] Navigation algorithms tested
- [ ] Safety systems functional

Network Verification

- [ ] Web interface loads correctly
- [ ] Real-time communication established
- [ ] No significant latency in control loop
- [ ] Connection recovery works properly

7.2 Operational Testing

Navigation Testing

```
# Test sequence
ros2 launch material_handling_robot full_system.launch.py

# Test web interface connection
# Navigate to http://ROBOT_IP:8080

# Test basic movement
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist '{linear: {x: 0.2}}'

# Test emergency stop
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist '{linear: {x: 0.0}, angular: {z: 0.0}}'
```

Material Handling Testing

1. **Position Accuracy**: Test navigation to each location
2. **Pickup Simulation**: Verify material count updates
3. **Job Execution**: Run complete pickup/dropoff cycles
4. **Queue Management**: Test multiple job handling
5. **Error Recovery**: Test connection loss scenarios

8. Maintenance and Monitoring

8.1 System Monitoring

Health Check Script

```
#!/usr/bin/env python3
# health_check.py
import subprocess
import psutil
import time

def check_system_health():
  health_report = {
    'timestamp': time.time(),
    'cpu_usage': psutil.cpu_percent(),
    'memory_usage': psutil.virtual_memory().percent,
    'disk_usage': psutil.disk_usage('/').percent,
    'ros_nodes': check_ros_nodes(),
    'network_latency': check_network_latency()
  }

  return health_report

def check_ros_nodes():
  try:
    result = subprocess.run(['ros2', 'node', 'list'], capture_output=True, text=True)
    return len(result.stdout.strip().split('\n'))
  except:
    return 0

def check_network_latency():
  try:
    result = subprocess.run(['ping', '-c', '1', 'localhost'], capture_output=True, text=True)
    # Parse ping time
    return 0.0  # Simplified
  except:
    return -1

# Run health check every 60 seconds
while True:
  health = check_system_health()
  print(f"System Health: CPU={health['cpu_usage']:.1f}%, 
Memory={health['memory_usage']:.1f}%, Nodes={health['ros_nodes']}")
  time.sleep(60)
```

8.2 Maintenance Schedule

Daily Checks

- [ ] Battery level and charging status
- [ ] Motor temperature and performance
- [ ] Sensor cleanliness and calibration
- [ ] System log review

Weekly Maintenance

- [ ] Encoder calibration verification
- [ ] Emergency stop system test
- [ ] Network security updates
- [ ] Performance data analysis

Monthly Maintenance

- [ ] Full system backup
- [ ] Hardware inspection
- [ ] Software updates
- [ ] Safety system recertification

9. Troubleshooting Guide

9.1 Common Issues

Connection Problems

Symptom: Web interface shows "DISCONNECTED" **Solutions**:

1. Check ROSBridge server status: ps aux | grep rosbridge
2. Verify port accessibility: netstat -tlnp | grep 9090
3. Check firewall settings: sudo ufw status
4. Restart ROSBridge: ros2 launch rosbridge_server rosbridge_websocket_launch.xml

Navigation Issues

Symptom: Robot doesn't move or moves incorrectly **Solutions**:

1. Check motor controller connection
2. Verify encoder readings: ros2 topic echo /odom
3. Test manual control: ros2 topic pub /cmd_vel geometry_msgs/msg/Twist '{linear: {x: 0.1}}'
4. Calibrate wheel parameters

Performance Issues

Symptom: Slow response or high latency **Solutions**:

1. Check system resources: htop
2. Optimize ROS2 QoS settings
3. Reduce web interface update rate

4. Check network bandwidth

9.2 Log Analysis

ROS2 Logging

```
# Enable debug logging
export RCUTILS_LOGGING_SEVERITY=DEBUG
ros2 launch material_handling_robot full_system.launch.py

# View specific node logs
ros2 run rqt_console rqt_console
```

System Logs

```
# Check system logs
journalctl -u ros2-material-handling.service -f

# Check hardware interface logs
tail -f ~/.ros/log/latest/material_handling_hardware.log
```

10. Performance Optimization

10.1 Real-time Performance

RT Kernel Installation

```
# Install RT kernel for better real-time performance
sudo apt install linux-lowlatency
sudo reboot
```

Process Priority

```
# Run critical nodes with higher priority
sudo chrt -f 50 ros2 run material_handling_robot hardware_interface
```

10.2 Network Optimization

QoS Configuration

```
# Optimize QoS for real-time control
from rclpy.qos import QoSProfile, QoSReliabilityPolicy, QoSHistoryPolicy

qos_profile = QoSProfile(
    reliability=QoSReliabilityPolicy.RELIABLE,
    history=QoSHistoryPolicy.KEEP_LAST,
    depth=1
)

self.cmd_vel_pub = self.create_publisher(Twist, 'cmd_vel', qos_profile)
```

This guide provides the foundation for implementing the material handling system with real hardware. Each section can be expanded based on specific hardware choices and operational requirements.