ACSAR Robot Hardware Interface Analysis: A Comprehensive Technical Documentation

Hardware Interface Code Analysis

June 9, 2025

Contents

1 Overview

1.1 Purpose and Scope

The ACSAR Hardware Interface serves as a critical bridge between ROS 2 control systems and a four-wheeled differential drive robot. This interface implements the ROS 2 `hardware_interface::SystemInterface` to provide real-time control and feedback for a mobile robot equipped with:

- Four independent wheels (front-left, front-right, rear-left, rear-right)
- Serial communication interface (RS-232/USB) to microcontroller
- LiDAR sensor for navigation and mapping
- Odometry calculation and publishing capabilities
- Transform broadcasting for robot localization

1.2 Hardware Architecture

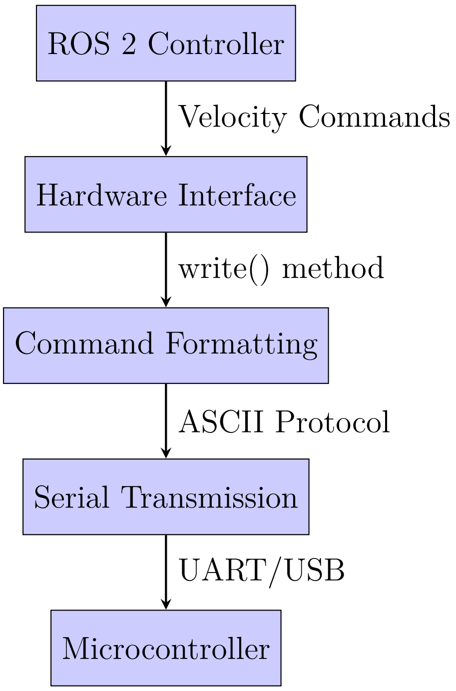
The system architecture consists of three main layers:

1. **ROS 2 Control Layer**: High-level motion planning and control
2. **Hardware Interface Layer**: This code - translates ROS commands to hardware protocols
3. **Microcontroller Layer**: Low-level motor control and sensor reading

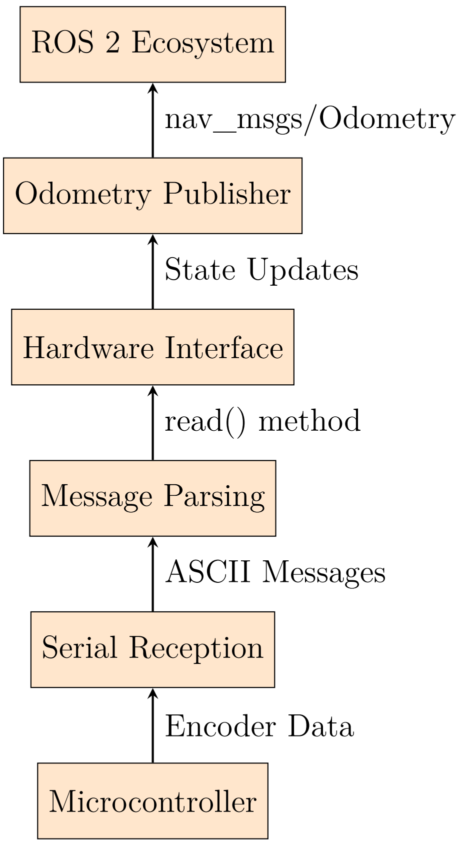
The communication protocol uses a custom ASCII-based message format over serial communication at configurable baud rates (9600-115200 bps).

2 Workflow

2.1 Data Flow: Software to Hardware



2.2 Data Flow: Hardware to Software



2.3 Command Protocol Format

The serial communication uses a comma-separated format:

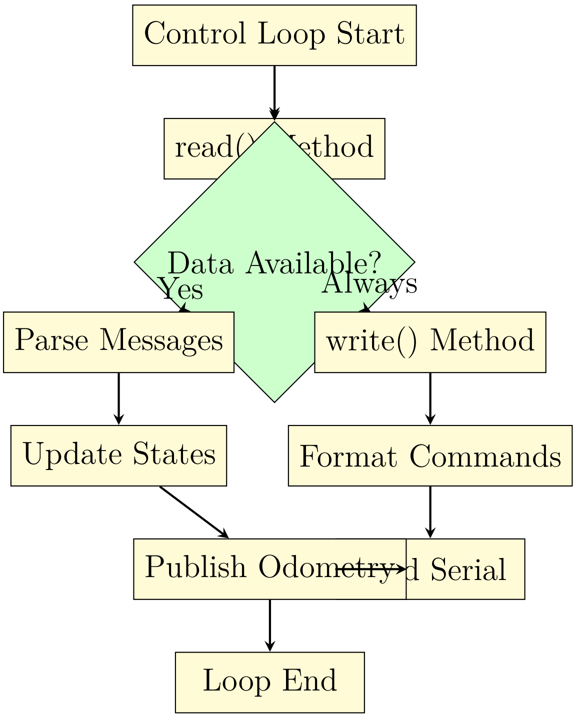
```
// Command Format: [motor_id][direction][velocity]
// Example: "ap12.50,bn08.75,cp15.00,dn10.25\r\n"
// Where: a=front_right, b=front_left, c=rear_left, d=rear_right
//        p=positive, n=negative
//        XX.XX = velocity value with 2 decimal places
```

3 Logical Flow

3.1 Initialization Sequence

1. **Constructor**: `ACSARInterface()` - Initialize empty object
2. **Parameter Loading**: `on_init()` - Extract configuration parameters
3. **Interface Export**: Export state and command interfaces
4. **Activation**: `on_activate()` - Open serial port and initialize publishers

3.2 Runtime Control Loop



3.3 Error Handling Logic

The interface implements comprehensive error handling:

- **Serial Communication Errors**: Try-catch blocks around all serial operations
- **Parameter Validation**: Default values for missing configuration parameters

- **Message Parsing**: Robust parsing with validation and error recovery
- **Resource Management**: Proper cleanup in destructor and deactivation

4 Mathematical Equations and Odometry Calculations

4.1 Wheel Velocity Calculation

Converting angular velocities to linear velocities for each wheel:

$$v_{\text{wheel}} = \omega_{\text{wheel}} \cdot r \tag{1}$$

Where:

- v_{wheel}: Linear velocity of the wheel (m/s)
- ω_{wheel}: Angular velocity from encoder data (rad/s)
- r: Wheel radius (m), configured via `wheel_radius_`

This equation is applied to each wheel (front-right, front-left, rear-left, rear-right) to compute their linear velocities:

```
double front_right_velocity = velocity_states_[0] * wheel_radius_;
double front_left_velocity = velocity_states_[1] * wheel_radius_;
double rear_left_velocity = velocity_states_[2] * wheel_radius_;
double rear_right_velocity = velocity_states_[3] * wheel_radius_;
```

4.2 Differential Drive Kinematics

Calculating the robot's linear and angular velocities:

$$v_x = \frac{v_{\text{left}} + v_{\text{right}}}{2} \tag{2}$$

$$\omega_z = \frac{v_{\text{right}} - v_{\text{left}}}{L} \tag{3}$$

Where:

- v_x: Linear velocity of the robot along the x-axis (m/s)
- v_{left}: Average linear velocity of left wheels (m/s)
- v_{right}: Average linear velocity of right wheels (m/s)
- ω_z: Angular velocity around the z-axis (rad/s)
- L: Wheel separation distance (m), configured via `wheel_separation_`

Implementation in code:

```
double left_wheel_velocity = (front_left_velocity + rear_left_velocity) /
    2.0;
double right_wheel_velocity = (front_right_velocity + rear_right_velocity)
    / 2.0;
double linear_vel_x = (left_wheel_velocity + right_wheel_velocity) / 2.0;
double angular_vel_z = (right_wheel_velocity - left_wheel_velocity) /
    wheel_separation_;
```

4.3 Position Update

Updating the robot's pose (x, y, θ) using numerical integration:

$$\Delta x = v_x \cdot \cos(\theta) \cdot \Delta t \tag{4}$$

$$\Delta y = v_x \cdot \sin(\theta) \cdot \Delta t \tag{5}$$

$$\Delta \theta = \omega_z \cdot \Delta t \tag{6}$$

$$x = x + \Delta x \tag{7}$$

$$y = y + \Delta y \tag{8}$$

$$\theta = \theta + \Delta \theta \tag{9}$$

Where:

- $\Delta x, \Delta y$: Change in position (m)
- $\Delta \theta$: Change in orientation (rad)
- Δt: Time step (s), derived from `period.seconds()`
- x, y: Robot's position in the od proceduresometry frame (m)
- θ: Robot's orientation (rad)

Implementation in code:

```
double dt = period.seconds();
double delta_x = linear_vel_x * std::cos(theta_) * dt;
double delta_y = linear_vel_x * std::sin(theta_) * dt;
double delta_theta = angular_vel_z * dt;
x_ += delta_x;
y_ += delta_y;
theta_ += delta_theta;
```

4.4 Orientation Normalization

Normalizing the orientation angle to keep it within $[-\pi, \pi]$:

$$\theta = \begin{cases} \theta - 2\pi, & \text{if } \theta > \pi \\ \theta + 2\pi, & \text{if } \theta < -\pi \\ \theta, & \text{otherwise} \end{cases} \tag{10}$$

Implementation in code:

```
while (theta_ > M_PI) theta_ -= 2 * M_PI;
while (theta_ < -M_PI) theta_ += 2 * M_PI;
```

4.5 Quaternion Conversion

Converting the orientation angle to a quaternion for transform broadcasting:

$$q = \begin{bmatrix} 0 \\ 0 \\ \sin\left(\frac{\theta}{2}\right) \\ \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{11}$$

Where:

- q: Quaternion $[x, y, z, w]$
- θ: Robot's orientation angle (rad)

Implementation in code using tf2:

```
tf2::Quaternion q;
q.setRPY(0, 0, theta_);
```

5 Behind the Scenes

5.1 Serial Communication Layer

5.1.1 LibSerial Library Integration

The interface uses the LibSerial library for cross-platform serial communication:

```
AVR_.SetBaudRate(LibSerial::BaudRate::BAUD_115200);
AVR_.SetCharacterSize(LibSerial::CharacterSize::CHAR_SIZE_8);
AVR_.SetParity(LibSerial::Parity::PARITY_NONE);
AVR_.SetStopBits(LibSerial::StopBits::STOP_BITS_1);
AVR_.SetFlowControl(LibSerial::FlowControl::FLOW_CONTROL_NONE);
```

5.1.2 Buffer Management

- **Input Buffers**: LibSerial manages receive buffers automatically
- **Output Buffers**: Commands are formatted and transmitted immediately
- **Buffer Flushing**: `FlushIOBuffers()` clears stale data during initialization

5.2 Memory Management

5.2.1 Dynamic Memory Allocation

- **Vector Resizing**: State and command vectors resize based on joint count
- **Smart Pointers**: ROS 2 node and publisher use shared_ptr for automatic cleanup
- **Thread Management**: Executor thread with proper join() in destructor

5.2.2 Hardware Register Interaction

While this code doesn't directly access hardware registers, the underlying LibSerial library interfaces with:

- **UART Registers**: Baud rate, parity, stop bits configuration
- **System Buffers**: Kernel-level serial buffers for data queuing
- **File Descriptors**: Linux/Unix file system interface to serial devices

5.3 Microcontroller Communication

5.3.1 Hardware Response Timing

When the interface sends commands, the microcontroller:

1. Receives UART interrupt
2. Parses command string
3. Updates motor PWM signals
4. Reads encoder feedback
5. Formats response message
6. Transmits encoder data back

5.3.2 Encoder Feedback Processing

The microcontroller sends encoder data in the format:

```
ap15.23,bn12.45,cp14.67,dn13.89
```

6 Call Hierarchy

6.1 Runtime Call Sequence

1. **ROS 2 Control Manager** calls `read()` at control frequency (typically 100Hz)
2. `read()` method:
    - Checks `AVR_.IsDataAvailable()`

  - Calls `AVR_.ReadLine()` if data present
  - Parses message using `std::stringstream`
  - Updates `velocity_states_` and `position_states_`
  - Calls `publishOdometry()`

3. **ROS 2 Control Manager** calls `write()` with new commands
4. `write()` method:
  - Formats commands using `std::stringstream`
  - Calls `AVR_.Write()` to transmit

6.2 Odometry Publishing Call Chain

```
publishOdometry()
 Calculate wheel velocities
 Compute differential drive kinematics
 Update robot pose (x, y, theta)
 Create geometry_msgs::TransformStamped
 tf_broadcaster_->sendTransform()
 Create nav_msgs::Odometry message
 odom_publisher_->publish()
 Publish LiDAR transform
```

7 Timing and Synchronization

7.1 Control Loop Timing

7.1.1 Real-Time Constraints

- **Control Frequency**: Typically runs at 100Hz (10ms cycle time)
- **Serial Communication**: Asynchronous, non-blocking reads
- **Odometry Publishing**: Synchronized with control loop
- **Transform Broadcasting**: Real-time requirements for navigation

7.1.2 Timing Mechanisms

1. **Polling-Based Reading**:

```
if (AVR_.IsDataAvailable()) {
    AVR_.ReadLine(message);
}
```

2. **Time-Based Integration**:

```
auto dt = (rclcpp::Clock().now() - last_read_time_).seconds();
position_states_[i] += velocity_states_[i] * dt;
```

3. **Executor Thread Management**:

```
executor_thread_ = std::thread([this]() {
    while (rclcpp::ok()) {
        executor_->spin_some();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
});
```

7.2 Synchronization Strategies

7.2.1 Data Consistency

- **Atomic Operations**: Single-threaded execution for state updates
- **Message Ordering**: Serial communication ensures command/response ordering
- **Time Stamping**: All messages timestamped with ROS 2 time

7.2.2 Thread Synchronization

- **Main Thread**: Handles read/write operations
- **Executor Thread**: Manages ROS 2 publishers and transforms
- **Synchronization**: Shared data protected by single-threaded access pattern

7.3 Latency Analysis

7.3.1 Communication Latency

- **Serial Transmission**: Depends on baud rate and message length
- **Microcontroller Processing**: Typically 1-5ms for command execution
- **Encoder Reading**: Hardware-dependent, usually sub-millisecond
- **Total Round-Trip**: Typically 5-10ms for command to feedback

7.3.2 Control System Latency

- **ROS 2 Control Loop**: 10ms base cycle time
- **Message Processing**: Sub-millisecond for parsing
- **Odometry Calculation**: Computational overhead minimal
- **Transform Broadcasting**: Real-time, no buffering

8 Configuration and Parameters

8.1 Hardware Parameters

- `serial_port`: Device path (e.g., "/dev/ttyUSB0")
- `baud_rate`: Communication speed (9600-115200)
- `wheel_radius`: Physical wheel radius in meters
- `wheel_separation`: Distance between wheel centers
- `odom_frame`, `base_frame`, `lidar_frame`: TF frame names
- `lidar_x_offset`, `lidar_y_offset`, `lidar_z_offset`: LiDAR mounting position

8.2 Performance Tuning

- **Baud Rate**: Higher rates reduce communication latency
- **Message Frequency**: Balance between responsiveness and CPU usage
- **Covariance Tuning**: Adjust odometry uncertainty based on robot characteristics
- **Wheel Parameters**: Accurate values critical for odometry accuracy

9 Conclusion

The ACSAR Hardware Interface represents a well-architected bridge between ROS 2 control systems and low-level hardware. Key strengths include:

- Robust error handling and resource management
- Flexible parameter configuration
- Real-time performance with proper timing management
- Comprehensive odometry and transform publishing
- Clean separation of concerns between control and communication layers

The interface successfully abstracts the complexity of serial communication and hardware control while providing the necessary interfaces for ROS 2 integration, making it suitable for research and development applications in mobile robotics.