Hardware Interface Code Analysis: Four-Motor Robot Control System

Embedded Systems Analysis

June 9, 2025

Contents

1 Overview

This C code implements a comprehensive hardware interface for a four-motor robotic system based on an AVR microcontroller. The system serves as a bridge between high-level robot control commands and low-level hardware actuation.

1.1 Primary Purpose

The code implements a real-time motor control system that:

- Receives velocity commands via UART communication
- Controls four independent DC motors with PWM speed control
- Monitors motor rotation using quadrature encoders
- Implements closed-loop PID control for precise velocity regulation
- Reports actual velocities back to the host system

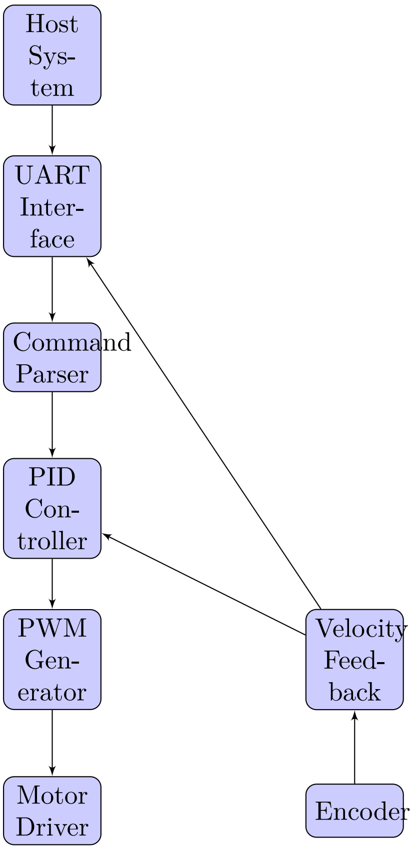
1.2 Hardware Components

The system interfaces with the following hardware:

- **Microcontroller**: AVR (8MHz clock)
- **Motors**: 4 DC motors with H-bridge drivers
- **Encoders**: 4 quadrature encoders (3828 pulses per revolution)
- **PWM Generators**: 3 hardware timers (Timer0, Timer1, Timer2)
- **Communication**: UART interface (9600 baud)
- **Interrupts**: External interrupts for encoder feedback

2 Workflow

2.1 High-Level Data Flow



2.2 Step-by-Step Workflow

1. **Command Reception**: Host system sends velocity commands via UART
2. **Command Parsing**: Incoming characters are buffered and parsed when complete
3. **Target Setting**: Parsed velocities are stored as target values for each motor
4. **Encoder Monitoring**: Interrupt-driven encoder counting tracks actual motor rotation
5. **Velocity Calculation**: Encoder counts are converted to angular velocities
6. **PID Control**: Error between target and actual velocities drives PID computation
7. **PWM Generation**: PID output is converted to PWM duty cycle for motor control
8. **Motor Actuation**: H-bridge drivers receive PWM signals and direction commands
9. **Feedback Transmission**: Actual velocities are reported back to host system

3 Logical Flow

3.1 Main Program Structure

```
int main(void) {
    _delay_ms(2000);
    init_system();
    sei();  // Enable global interrupts

    while (1) {
```

```c
        // UART command processing
        if (UART_ReceivePeriodic(&c) != 0) {
            // Buffer incoming characters
            // Parse complete commands
        }

        // Periodic motor control update
        if ((system_ticks - last_control_update >= 50)) {
            update_motors();
            last_control_update = system_ticks;
        }
    }
}
```

Listing 1: Main Program Loop

3.2 Control Flow Breakdown

3.2.1 Initialization Sequence

1. Digital I/O configuration
2. UART initialization (9600 baud)
3. Timer setup for PWM generation
4. External interrupt configuration for encoders
5. Motor direction and speed initialization

3.2.2 Command Processing Logic

```c
void process_command(void) {
    // Parse format: a[p/n]XX.XX,b[p/n]XX.XX,c[p/n]XX.XX,d[p/n]XX.XX
    char *ptr = cmd_buffer;

    while (*ptr) {
        char motor_id = *ptr++;           // Extract motor identifier
        char direction = *ptr++;          // Extract direction (p/n)
        // Extract velocity value
        // Convert to target velocity
        // Store in target_velocities array
    }
}
```

Listing 2: Command Processing Flow

3.2.3 Motor Control Update Logic

```c
void update_motors(void) {
    // Calculate actual velocities from encoder counts
    for (u8 i = 0; i < 4; i++) {
        double new_velocity = (encoder_counts[i] * (1000.0 / CONTROL_INTERVAL_MS)) *
            RAD_PER_SEC;
        // Apply low-pass filtering
        measured_velocities[i] = alpha * new_velocity + (1.0 - alpha) * measured_velocities[i
            ];
    }

    // Apply PID control
    for (u8 i = 0; i < 4; i++) {
        if (absolute(target_velocities[i]) > 0.1) {
            set_motor_direction(i, target_velocities[i] >= 0);
            motor_commands[i] = calculate_pid(i);
        }
        set_motor_speed(i, motor_commands[i]);
    }

```

```
    // Reset encoder timing and report
}
```

Listing 3: Motor Control Update

4 Behind the Scenes

4.1 Hardware Timer Configuration

- **Timer0**: Generates PWM for Motor C and system tick interrupt
- **Timer1**: Generates PWM for Motors A and B (16-bit timer, dual output)
- **Timer2**: Generates PWM for Motor D

4.2 Register-Level Operations

```
void set_motor_speed(u8 motor_index, u8 speed) {
    switch(motor_index) {
        case 0: OCR1A = speed; break;  // Timer1 Output Compare A
        case 1: OCR1B = speed; break;  // Timer1 Output Compare B
        case 2: OCR0 = speed; break;   // Timer0 Output Compare
        case 3: OCR2 = speed; break;   // Timer2 Output Compare
    }
}
```

Listing 4: PWM Register Control

```
void set_motor_direction(u8 motor_index, Bool_t forward) {
    if (!forward) {
        Dio_WritePin(dir1_pin, HIGH);
        Dio_WritePin(dir2_pin, LOW);
    } else {
        Dio_WritePin(dir1_pin, LOW);
        Dio_WritePin(dir2_pin, HIGH);
    }
}
```

Listing 5: Direction Control

4.3 Interrupt Service Routines

```
void UpdateEncoder_Motor_a(void) {
    Bool_t phase_B = Dio_ReadPin(ENCODER_A_PHASE_B);

    if (phase_B) {
        encoder_counts[0]--;  // Reverse direction
    } else {
        encoder_counts[0]++;  // Forward direction
    }
}
```

Listing 6: Encoder ISR Example

```
void update_system_tick(void) {
    static u8 ms_counter = 0;
    if (++ms_counter >= 4) {
        system_ticks++;
        ms_counter = 0;
    }
}
```

Listing 7: System Tick Handler

4.4 Memory-Mapped I/O Operations

- **OCR0, OCR1A, OCR1B, OCR2**: Output Compare Registers for PWM duty cycle
- **PORTA, PORTD**: Digital output ports for motor direction control
- **PINB, PINC, PIND**: Digital input ports for encoder reading
- **UDR, UCSRA**: UART data and status registers

5 Call Hierarchy

5.1 Function Call Tree

The function call hierarchy is divided into three sub-hierarchies to improve clarity, each focusing on a specific aspect of the system's operation.

5.1.1 Initialization Sub-Hierarchy

This sub-hierarchy details the initialization process executed at system startup.

init_system()

DIO_Init() UART_Init() TIMER_Init() EXI_Setup()

5.1.2 Command Processing Sub-Hierarchy

This sub-hierarchy covers the processing of incoming UART commands.

process_command()

atof()

5.1.3 Motor Control Sub-Hierarchy

This sub-hierarchy outlines the motor control update process, including PID control and motor actuation.

update_motors()

calculate_pid() set_motor_direction() set_motor_speed() report_velocities()

5.2 Interrupt-Driven Call Hierarchy

- **External Interrupts**:
  - INT0 → UpdateEncoder_Motor_a()
  - INT1 → UpdateEncoder_Motor_b()
  - INT2 → UpdateEncoder_Motor_c()
- **Timer Overflow Interrupt**:
  - Timer0 Overflow → update_system_tick()

6 Timing and Synchronization

6.1 Real-Time Constraints

- **Control Loop Frequency**: 20 Hz (50ms interval)
- **System Tick**: 1 ms resolution
- **PWM Frequency**: Determined by timer prescaler and TOP value
- **Encoder Sampling**: Interrupt-driven, real-time response

6.2 Synchronization Mechanisms

```
if ((system_ticks - last_control_update >= 50)) {
    update_motors();
    last_control_update = system_ticks;
}
```

Listing 8: Software Timer Check

- **Encoder Interrupts**: Provide immediate response to rotation changes
- **Global Interrupt Enable**: `sei()` enables all configured interrupts
- **Atomic Operations**: Encoder count updates are atomic due to ISR context

6.3 Timing Analysis

Operation	Frequency	Timing Source
System Tick	1 kHz	Timer0 Overflow
Control Loop	20 Hz	Software Timer
PWM Output	∼1 kHz	Hardware Timers
Encoder Reading	Event-driven	External Interrupts
UART Communication	9600 baud	Hardware UART

Table 1: System Timing Summary

6.4 Critical Sections

- **Encoder Count Access**: Shared between ISRs and main loop
- **Velocity Calculation**: Must be atomic to prevent inconsistent readings
- **Command Buffer**: Shared between UART reception and command processing

7 PID Control Implementation

7.1 Controller Structure

```
typedef struct {
    double kp;          // Proportional gain
    double ki;          // Integral gain
    double kd;          // Derivative gain
    double integral;    // Integral accumulator
    double prev_error;  // Previous error for derivative
} PIDController;
```

Listing 9: PID Controller Definition

7.2 Control Algorithm

```c
double calculate_pid(u8 motor_index, double target, double measured

) {
    PIDController *pid = &pid_controllers[motor_index];
    double error = target - measured;

    // Proportional term
    double p_term = pid->kp * error;

    // Integral term with windup protection
    pid->integral += error * (CONTROL_INTERVAL_MS/1000.0);
    pid->integral = CLAMP(pid->integral, -5.0, 5.0);
    double i_term = pid->ki * pid->integral;

    // Derivative term
    double d_term = pid->kd * (error - pid->prev_error) * (1000.0 / CONTROL_INTERVAL_MS);
    pid->prev_error = error;

    return CLAMP(p_term + i_term + d_term, 0.0, 255.0);
}
```

Listing 10: PID Calculation

8 Communication Protocol

8.1 Command Format

The system uses a custom ASCII protocol for velocity commands:

`a[p/n]XX.XX,b[p/n]XX.XX,c[p/n]XX.XX,d[p/n]XX.XX`

Where:

- `a,b,c,d`: Motor identifiers
- `p/n`: Direction (positive/negative)
- `XX.XX`: Velocity value in rad/s

8.2 Response Format

Velocity feedback follows the same format as commands, transmitted at 20 Hz.

9 Conclusion

This hardware interface code implements a sophisticated real-time motor control system that effectively bridges high-level robot control commands with low-level hardware actuation. The system demonstrates proper use of embedded systems principles including interrupt-driven programming, real-time control loops, and hardware abstraction layers.

The modular design allows for easy maintenance and modification, while the PID control implementation ensures precise velocity regulation. The communication protocol provides a simple yet effective interface for higher-level control systems.