Joystick-Controlled Robot Car

Using HCS12 Microcontroller and MATLAB Simulink

A Complete Embedded Systems Project

Integrating Analog Joystick Control, Serial Communication, Motor Drivers, and Model-Based Design

---

Project Team:

- Tasneem Barakat
- Malak Alhajaj
- Renad Al-Nimat

Project Type: Embedded Systems / Robotics
Controller: Dragon12-JR (MC9S12DG256)
Motor Driver: Arduino Uno + L298N (Simulink)
Communication: Serial UART (9600 baud)
Date: January 2026

Abstract

This report presents the design and implementation of a joystick-controlled robot car that demonstrates the integration of multiple embedded systems working together. The project uses a Dragon12-JR development board based on the Freescale MC9S12DG256 microcontroller to read analog joystick inputs and transmit movement commands via serial communication to an Arduino Uno. The Arduino, programmed using MATLAB Simulink model-based design, controls four DC motors through an L298N H-bridge motor driver, enabling the robot to move forward, backward, turn left, and turn right.

The project showcases practical applications of analog-to-digital conversion, serial communication protocols, PWM-based motor speed control, and real-time embedded programming. A key highlight of this project is the use of MATLAB Simulink for the motor controller implementation, demonstrating the power of model-based design in embedded systems development. This approach allowed us to visually design the control logic using blocks and automatically generate deployable code for the Arduino platform, eliminating the need for traditional hand-written code.

The system successfully implements five movement commands: Forward, Backward, Left, Right, and Stop, with visual feedback provided through a 7-segment display on the

Dragon12-JR board. The modular architecture separates the control logic (HCS12) from the motor driving logic (Arduino/Simulink), making the system easier to develop, test, and maintain.

1. Introduction and Background

1.1 Project Overview

The goal of this project was to build a remotely controlled robot car that could be operated using a simple joystick interface. Rather than using wireless communication like Bluetooth or WiFi, we chose to use a wired serial connection to focus on the fundamentals of embedded communication and motor control. This approach allowed us to understand the low-level details of how microcontrollers communicate with each other and how motor drivers work.

The project is divided into two main parts: the controller side and the driver side. On the controller side, we have the Dragon12-JR board that reads the joystick position and decides what command to send. On the driver side, we have an Arduino Uno that receives these commands and controls the motors accordingly. The Arduino was programmed entirely using MATLAB Simulink's model-based design approach, which provided a visual and intuitive way to implement the motor control logic.

1.2 Project Objectives

When we started this project, we set out to achieve several learning objectives:

- Gain hands-on experience with the HCS12 microcontroller family, which is widely used in automotive and industrial applications
- Understand how analog signals from sensors like joysticks are converted to digital values and processed by microcontrollers
- Implement reliable serial communication between two different microcontroller platforms
- Explore model-based design using MATLAB Simulink as an alternative to traditional hand-coded programming
- Learn PWM-based motor speed control and H-bridge motor driver operation

1.3 Background on Model-Based Design

Model-based design is an approach where engineers create a visual model of their system using blocks and connections, and then automatically generate code from that model. MATLAB Simulink is one of the most popular tools for this approach, especially in automotive and aerospace industries. For our project, we created a Simulink model that implements all the motor control functionality, which could then be automatically deployed to the Arduino hardware.

The advantages of using Simulink for this project include: visual representation that makes the system easier to understand at a glance, ability to simulate the model before deploying to hardware, easy parameter modification through block properties, and automatic code generation that ensures consistency between the model and the actual code running on the hardware.

2. System Design

2.1 Mechanical Design

The robot car is built on a standard 4WD chassis with four DC motors, one for each wheel. The chassis provides a stable platform for mounting all electronic components including the L298N motor driver, Arduino Uno, and battery pack. The four-wheel drive configuration provides good traction and allows for differential steering, where the car turns by running the wheels on each side at different speeds or directions.

The motors are arranged with two motors on each side of the chassis. Both motors on the same side are connected in parallel to the same H-bridge channel, allowing them to be controlled together. This configuration simplifies the control logic while providing adequate power and stability. The left-side motors are connected to Channel A (outputs OUT1 and OUT2), while the right-side motors are connected to Channel B (outputs OUT3 and OUT4).

2.2 Electrical Design

2.2.1 System Architecture

Our robot car system follows a master-slave architecture where the HCS12 acts as the master controller and the Arduino acts as the slave driver. The data flow in our system is: the joystick produces analog voltages representing its position, these voltages are read by the HCS12's analog-to-digital converter, the HCS12 compares these values against thresholds to determine the desired direction, and based on this decision, it sends a single character command over the serial connection to the Arduino.

Table 1: Command Protocol

Command	ASCII	Decimal	Action	7-Segment
'F'	0x46	70	Move Forward	4
'B'	0x42	66	Move Backward	3
'L'	0x4C	76	Turn Left	1
'R'	0x52	82	Turn Right	2
'S'	0x53	83	Stop	0

2.2.2 Hardware Components

Dragon12-JR Development Board: The Dragon12-JR is an educational development board built around the Freescale MC9S12DG256 microcontroller. This 16-bit microcontroller features 256 KB of flash memory, 12 KB of RAM, and 4 KB of EEPROM. It runs from an 8 MHz crystal with a 4 MHz default bus clock. The board includes a 7-segment display connected to Port H (common-anode configuration) which we used for visual feedback of the current command.

Analog Joystick: A standard two-axis analog joystick module with two potentiometers for X-axis (left-right) and Y-axis (up-down) control. Through calibration, we determined that our joystick's X-axis centers around 550 and Y-axis centers around 150 (different from the typical 512 center value).

Arduino Uno: Based on the ATmega328P microcontroller running at 16 MHz, the Arduino serves as the motor controller. It receives commands via serial communication and drives the motors using PWM signals. The Arduino was programmed using MATLAB Simulink model-based design.

L298N H-Bridge Motor Driver: A dual H-bridge motor driver that can control two DC motors independently. Channel A (ENA, IN1, IN2) drives the left-side motors via OUT1 and OUT2, while Channel B (ENB, IN3, IN4) drives the right-side motors via OUT3 and OUT4. The enable pins accept PWM signals for speed control.

Table 2: Arduino to L298N Wiring

Arduino Pin	L298N Pin	Function
Pin 8	IN1	Left motor direction
Pin 9	IN2	Left motor direction
Pin 11	IN3	Right motor direction
Pin 10	IN4	Right motor direction
Pin 6 (PWM)	ENA	Left motor speed
Pin 5 (PWM)	ENB	Right motor speed

2.3 Software Design

2.3.1 HCS12 Controller Software

The HCS12 software is written in C and follows a simple structure. After initializing all the peripherals (ADC, serial port, display, and button input), the program enters an infinite loop where it continuously reads the joystick, determines the command, and sends it to the Arduino. This polling approach is simple and works well for our application.

Analog-to-Digital Conversion: Reading the joystick involves using the ATD1 module. The joystick is connected to PAD8 (X-axis) and PAD9 (Y-axis), which are inputs to ATD1. The initialization process involves powering up the ATD module by setting the ADPU bit in

ATD1CTL2, configuring for 10-bit resolution, and then performing conversions by writing to ATD1CTL5 and waiting for the SCF flag. The ADC read function is implemented as:

```
unsigned int adc_read(unsigned char ch) {
    ATD1CTL5 = 0x20 | (ch & 0x0F);
    while(!(ATD1STAT0 & 0x80));
    return ATD1DR0 & 0x03FF;
}
```

Joystick Calibration: Through careful testing, we calibrated the joystick thresholds. The X-axis centers around 550 with left threshold at 350 and right threshold at 750 (deadzone of ±200). The Y-axis centers around 150 with forward threshold at 350. Due to the Y-axis being partially non-functional for the upward direction, we implemented a push button on PA0 for the backward command.

Table 3: Joystick Calibration Values

Axis	Center Value	Low Threshold	High Threshold
X (Left-Right)	~550	350 (Left)	750 (Right)
Y (Forward-Back)	~150	50 (Broken)	350 (Forward)

Command Processing Logic: The get_command() function determines which command to send based on joystick position and button state. The push button is checked first (highest priority for backward), followed by X-axis thresholds for left/right, then Y-axis threshold for forward. If no direction is detected, the stop command is returned:

```
char get_command(unsigned int x, unsigned int y) {
    if(button_pressed()) return 'B';  // Backwards priority
    if(x <= LEFT_TH)  return 'L';
    if(x >= RIGHT_TH) return 'R';
    if(y >= DOWN_TH)  return 'F';
    return 'S';  // Deadzone = Stop
}
```

Serial Communication: The HCS12 uses SCI1 for communication with the Arduino. We configure SCI1 with SCI1BDL = 26 for 9600 baud at 4 MHz bus clock (formula: SCI1BD = Bus Clock / (16 × Baud Rate) = 4,000,000 / 153,600 ≈ 26). The serial_send() function waits for the TDRE flag before transmitting each character. Commands are sent continuously at approximately 20 times per second (every 50ms delay).

7-Segment Display: The display provides visual feedback by showing the current command: 0 for Stop, 1 for Left, 2 for Right, 3 for Backward, and 4 for Forward. Since the display uses common-anode configuration, we invert the segment patterns in the code. The display is connected to Port H and updated whenever the command changes.

2.3.2 MATLAB Simulink Model-Based Design

For the motor controller on the Arduino, we employed MATLAB Simulink's model-based design approach instead of traditional hand-written code. This method allows us to visually

design the control logic using interconnected blocks, simulate the behavior before deployment, and automatically generate code for the target hardware.

Model Architecture: The Simulink model consists of the following key components:

- **Serial Receive Block:** Configured for Port 0 at 9600 baud, this block receives the ASCII command characters from the HCS12.
- **Command Comparators:** Five Relational Operator blocks compare the received command against constants (70, 66, 76, 82, 83) to detect Forward, Backward, Left, Right, and Stop commands respectively.
- **Logic OR Gates:** Logical Operator blocks combine the comparison results to determine motor direction outputs. For example, IN1 = (Forward OR Right), IN2 = (Backward OR Left), etc.
- **Data Type Converters:** Convert boolean outputs to uint8 format required by the Arduino Digital Output blocks.
- **Speed Control Switches:** Switch blocks select between the speed value (80) when moving and zero when stopped, based on the Enable signal (any movement command active).
- **Arduino Digital Output Blocks:** Four blocks for motor direction control (Pins 8, 9, 10, 11).
- **Arduino PWM Blocks:** Two blocks for motor speed control (Pins 5, 6).

Table 4: Motor Control Logic

Movement	IN1	IN2	IN3	IN4	ENA/ENB
Forward	HIGH	LOW	HIGH	LOW	80
Backward	LOW	HIGH	LOW	HIGH	80
Turn Left	LOW	HIGH	HIGH	LOW	80
Turn Right	HIGH	LOW	LOW	HIGH	80
Stop	LOW	LOW	LOW	LOW	0

Boolean Logic Implementation: The direction pins are determined by Boolean logic equations implemented using OR gates in Simulink:

- IN1 = isForward OR isRight (left motor forward when going forward or turning right)
- IN2 = isBackward OR isLeft (left motor backward when going backward or turning left)
- IN3 = isForward OR isLeft (right motor forward when going forward or turning left)
- IN4 = isBackward OR isRight (right motor backward when going backward or turning right)

The Enable signal is computed as: Enable = isForward OR isBackward OR isLeft OR isRight, which is false only when the Stop command is received.

Deployment Process: To deploy the Simulink model to the Arduino, we use the Simulink Support Package for Arduino Hardware. The model settings are configured for Arduino Uno as the target hardware with a fixed-step discrete solver and 0.05 second sample time. The deployment process automatically converts the Simulink blocks into C code, compiles it with the Arduino toolchain, and uploads the resulting binary to the board. This eliminates the need for manual coding and ensures consistency between the model design and the actual implementation.

Table 5: Simulink Model Block Summary

Block Type	Quantity	Purpose
Serial Receive	1	Receive commands from HCS12
Constant	7	Command codes (70,66,76,82,83), speed (80), zero
Relational Operator	5	Compare received command with constants
Logical Operator (OR)	5	Direction logic and enable signal
Data Type Conversion	4	Boolean to uint8 for Arduino
Switch	2	Speed selection (80 or 0)
Digital Output	4	Motor direction pins (8, 9, 10, 11)
PWM	2	Motor speed pins (5, 6)

3. Problems and Recommendations

3.1 Problems Encountered

During the development process, we encountered several challenges that taught us valuable lessons about embedded systems development:

1. Serial Communication Pin Confusion: Initially, we connected the wrong pins for serial communication. We used PS2 (which is the receive pin RXD1) instead of PS3 (the transmit pin TXD1) on the HCS12. This caused hours of debugging with no data being received by the Arduino. The solution was to carefully re-read the MC9S12DG256 datasheet and verify the pin functions. Lesson learned: Always verify pin assignments from the official datasheet before wiring.

2. Baud Rate Mismatch: When using the D-Bug12 monitor for debugging, the PLL is enabled and the bus runs at 24 MHz. But when running standalone, it runs at 4 MHz. Using the wrong value in our baud rate calculation (SCI1BDL) caused garbage characters to be received. The solution was to use SCI1BDL = 26 for standalone operation at 4 MHz bus clock.

3. Broken Joystick Y-Axis: We discovered that our joystick's Y-axis (up-down direction) was partially broken - the upward position wouldn't register properly (values never went

below 50). Instead of replacing the hardware, we adapted by adding a push button using the SW pin for the backward command. This taught us an important lesson about adaptability in engineering - sometimes you have to work around hardware limitations with software solutions.

4. Motors Going in Circles: During initial testing, the car would go in circles instead of moving straight. After investigation, we discovered that the motors on each side of the car were wired with opposite polarities, causing the front and back wheels on the same side to fight against each other. The solution was to ensure both motors on each side have the same polarity connection to the H-bridge outputs. In some cases, we also swapped IN3 and IN4 in the Simulink model to correct directional issues.

5. PWM Pin Compatibility: Initially, we attempted to use a non-PWM pin for motor speed control, which resulted in the motor always running at full speed regardless of the PWM value. The solution was to verify that Pins 5 and 6 on the Arduino are PWM-capable (connected to Timer 0) and use only those for the ENA and ENB connections.

Table 6: Issues and Solutions Summary

Issue	Symptom	Solution
Wrong serial pin	No data received	Use PS3 (TX), not PS2 (RX)
Wrong baud rate	Garbage characters	SCI1BDL=26 for 4MHz bus
Joystick Y-axis broken	No backward movement	Added push button for backward
Motors going in circles	Car not going straight	Fixed motor wiring polarity
Non-PWM pin used	Motor always full speed	Use PWM pins (5, 6)

3.2 Recommendations for Future Work

- **Wireless Communication:** Adding Bluetooth or WiFi communication would eliminate the need for a wired connection, making the system more practical for real-world applications.
- **Obstacle Avoidance:** Adding ultrasonic distance sensors could enable automatic obstacle detection and avoidance capabilities.
- **Proportional Speed Control:** Implementing proportional control based on joystick position (instead of just on/off) would provide more nuanced speed control for smoother operation.
- **Better Joystick:** Using a higher-quality joystick with reliable operation on both axes would eliminate the need for the backup push button.
- **Battery Monitoring:** Adding battery voltage monitoring to warn when power is low would prevent unexpected shutdowns.
- **Camera Integration:** Adding a camera and implementing computer vision could enable autonomous navigation and line-following capabilities.

4. Conclusion

This project successfully demonstrated the integration of multiple embedded systems to create a functional joystick-controlled robot car. We gained hands-on experience with the HCS12 microcontroller, analog-to-digital conversion, serial communication, PWM motor control, and most importantly, model-based design using MATLAB Simulink.

The use of MATLAB Simulink for the motor controller implementation proved to be an excellent choice. The visual block-based design made the control logic easy to understand, modify, and debug. The automatic code generation ensured consistency between our design and the actual implementation, while also eliminating potential coding errors that might occur with hand-written code. This experience has convinced us of the value of model-based design for embedded systems development.

The system uses a simple but effective command protocol where single ASCII characters represent movement commands. This approach is easy to implement, debug, and extend. The modular architecture separates the control logic (HCS12) from the motor driving logic (Arduino/Simulink), making each part easier to develop and test independently.

Perhaps the most valuable lesson from this project is the importance of careful debugging and systematic testing. Many hours were spent tracking down issues that turned out to have simple solutions, like using the wrong pin or having the wrong baud rate setting. Reading datasheets carefully and testing components individually before integration would have saved considerable time.

We also learned that hardware problems can often be solved in software. When the joystick's Y-axis proved unreliable, we added a button instead of replacing the hardware. When the motors ran in opposite directions, we adjusted the pin assignments in the Simulink model instead of rewiring. This flexibility is one of the great strengths of embedded systems and model-based design.

In conclusion, this project provided a comprehensive learning experience in embedded systems, from low-level hardware interfacing to high-level model-based design. The skills and knowledge gained will be invaluable for future projects in robotics, automation, and embedded systems development.

— End of Report —