자동주차 시스템을 적용한 RC 카 구동

Driving an RC Car with Automatic Parking System

황 시 환, 문 선 빈
(Sihwan Hwang[1], Sunbin Moon[1])
[1]School of Mechanical & Control Engineering, Handong Global University

Abstract: In this project, we have designed a RC car which traces the line and performs autonomous parking. We have used multi-sensor, multi-interrupt embedded system, with a goal of using as less sensors as possible. In our design, we have used only 3 sensors which are: 2 IR sensors for line tracing, and 1 ultrasonic sensor for distance measurement. We have used RC motor to rotate the direction of the ultrasonic sensor so that we can measure 3 sides of the car using only 1 ultrasonic sensor. Our focus was on stability and efficiency of line tracing; therefore, we were able to complete the track in 30 seconds which is 2nd in the section. In this paper, we have explained the setting of the system and algorithm in section 2 and section 3, respectively.

Keywords: Embedded system, embedded controller, ARM processor, multi-sensor

I. Introduction

In this project, we have designed an RC Car with automatic parking system. The objective of the vehicle is to drive the whole track by tracing the outer line, and parking safely at the end, with the minimum number of sensors. With our focus on design was stability and efficiency, we have designed the vehicle with 3 sensors total. Another objective is to complete the track in shortest time, where track is shown in Fig 1 below.

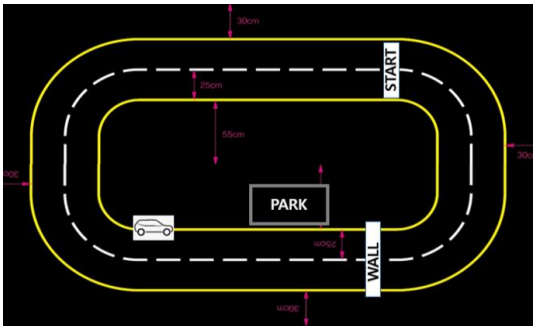


Fig 1. Track image

In order to design embedded system as such, we must follow the following procedure. First, set clock, peripherals, timers, and such. Then by using interrupt rather than polling, we can read data from multiple sensors without overloading the CPU. Finally, we can control the vehicle using these sensors and motors.

II. Methods

1. Port setting

Table 1 below shows the overall port settings. We have used total 3 sensors which are: 2 IR sensors and 1 ultrasonic sensor. In order to use only 1 UT sensor, we have used RC motor to rotate the UT sensor so that it can detect 3 sides as shown in Fig 2.

Table 1. Port setting

Sensor	Clock	GPIO-	Pin	Usage
Ultra Sonic	TIM3	GPIOC	8	Trigger
	TIM4	GPIOB	8	Echo
IR Sensor	TIM5	GPIOA	0	ADC
		GPIOA	1	ADC
RC Motor	TIM3	GPIOA	6	
Bluetooth		GPIOB	6	TX
		GPIOB	3	RX
DC motor	TIM1	GPIOA	8	Left High
	TIM1	GPIOA	9	Left Low
	TIM1	GPIOA	10	Right High
	TIM1	GPIOA	11	Right Low

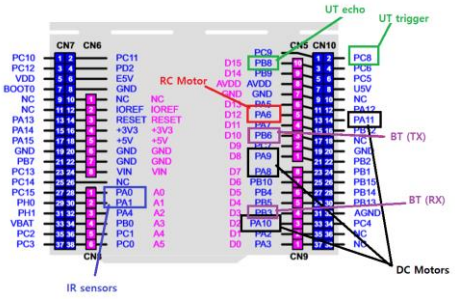


Fig 2. Port setting

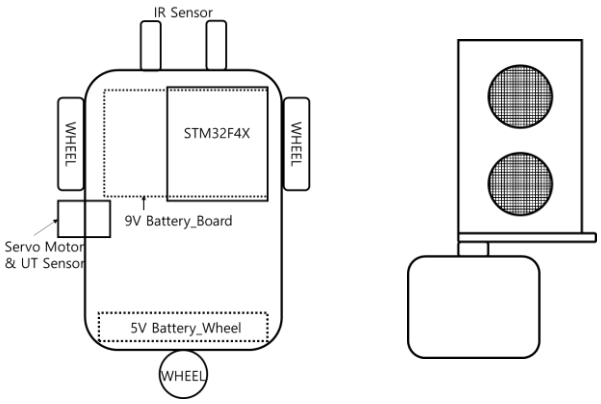


Fig 3. Vehicle Design with UT on RC motor

2. Setup

```
void setup_final(void)
{
    RCC_PLL_init();
    SysTick_init();
    EXTI_Init(2,13,1,0);
    UART2_init(9600, POL);

    BT_init(9600);            // BlueTooth

    RC_motor_setup();         // RC motor

    UT_setup();               // Ultrasonic Sensor

    DC_motor_setup();         // Wheels

    IR_setup();               // IR sensor

    TIM9_init(1);             // event handler
}
```

Fig 4. Setup

In the setup_final function, we setup and initialize everything from clock, timer, PWM and such. In 'RCC_PLL_init', it sets the 84 MHz PLL clock. Also, to manipulate the program, we added 'EXTI_init' for button input. 'UART2_init' was set so that we can use TerraTerm for debugging.

For Bluetooth communication, we set Bluetooth using 'BT_init' function. For the UT sensor on RC motor shown in Fig 3, we used 'RC_motor_setup' and 'UT_setup' function. The RC motor can be rotated by changing the pulse width of the PWM as table 2 below.

Table 2. RC motor control

Orientation	Pulse width [ms]
0° (front facing)	0.5
90° (side facing)	1.5
180° (rear facing)	2.5

For the ultrasonic sensor, we use two input (PC8, PB8) to get the result as shown in Fig 5 below.

```
void UT_setup(void)
{
    // trigger
    PWM_init(&pwm_UT,GPIOC,8);
    PWM_period_ms(&pwm_UT, 50);
    PWM_pulsewidth_ms(&pwm_UT,10);

    // echo
    CAP_init(&tim4_UT, GPIOB,8);
    TIM_period_us(4,10);
    NVIC_SetPriority(TIM4_IRQn, 0);
    NVIC_EnableIRQ(TIM4_IRQn);

}
```

Fig 5. UT setup

```
void TIM4_IRQHandler(void)  // UT read
{
    if(TIM4->SR & TIM_SR_UIF)
    {
        del++;//  <-- overflow counting
        TIM4->SR &= ~TIM_SR_UIF;
    }
    if((TIM4->SR & TIM_SR_CC3IF) != 0)  // rising edge
    {
        ris  = TIM4->CCR3;
        del = 0;
        TIM4->SR &= ~TIM_SR_CC3IF;
    }
    if((TIM4->SR & TIM_SR_CC4IF) != 0)  // falling edge
    {
        fal  = TIM4->CCR4;
        UT_distance = (double)(fal - ris + del*TIM4->ARR);
        UT_distance = UT_distance * (TIM4->PSC + 1)/84;
        UT_distance = UT_distance/58.0;
        del = 0;
        TIM4->SR &= ~TIM_SR_CC4IF;
    }
}
```

Fig 6. ultrasonic sensor read interrupt

To drive the wheel, we used 'DC_motor_setup' to set PWM input for motor driver. In order to control the wheel, we can decide the direction and speed by adjusting the duty of the 4 PWMs. For the IR input, we use Timer5 to trigger ADC. In order to make the line tracing as fast as possible, we have added the command in the ADC trigger. As shown in Fig 7, as soon as the ADC reads the IR value, it determines if it is detecting line or not, and changes the direction right away.

```
void ADC_IRQHandler(void){
  if((ADC1->SR & ADC_SR_OVR) == ADC_SR_OVR){
    ADC1->SR&=~ADC_SR_OVR;
  }

  if(ADC1->SR & ADC_SR_JEOC )
    {

        IR_left = ADC1->JDR1;
        IR_left = IR_left < IR_THRESH ? 1: 0;

        IR_right = ADC1->JDR2;
        IR_right = IR_right < IR_THRESH ? 1: 0;

        if(state == 1)
        {
            direction = !IR_left && IR_right ? 'r' : (IR_left && !IR_right ? 'l' : 's');
            drive(direction);
        }
        ADC1->SR &= ~(ADC_SR_JEOC);
    }
}
```

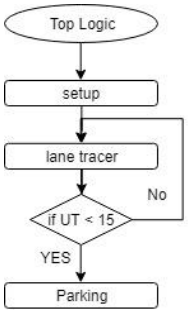
Fig 7. IR read & react

Finally, we use the timer 9 to increase the global variable ‘cnt' every 1ms, in order to use it as delay function in parking state. This will be explained in more detail at section 3.

III. Algorithm

1. Top logic

The flowchart of the overall algorithm is shown in Fig 8. We use the notion of state in process, starting from state 0 which is setup. Then, we take Bluetooth signal to go to state 1, which is lane tracer algorithm. After the vehicle is at 15cm distance to the wall, it goes to state 3, which is parking state.




```c
#include "manage.h"

int main()
{
    setup_final();
    printf("setup done!\r\n");
    BT_write((uint8_t *)"Hello Bluetooth\r\n", 17);

    while(1)
    {

        switch(state)
        {
            case 0: wait_BT(); break;
            case 1: lane_tracker(); break;
            case 2: park_vehicle(); break;

            case 9: sensor_testing(); break;
            default: break;
        }

    }
    drive('x');
}
```

Fig 8. Top logic Flowchart

2. Lane tracing logic

The lane tracing algorithm is very simple, so flowchart is needless. During lane tracing the vehicle goes straight, and every time IR sensors senses the lane, it interrupts and turns the vehicle so that the lane stays between the sensors.

```c
void lane_tracker(void)
{
    if(cnt < 500)
    {
        PWM_pulsewidth_ms(&pwm_RC, RC_front);
    }
    else if(UT_distance > 20 || UT_distance < 0)
    {
        drive(direction);
    }
    else
    {
        drive('X');  // stop vehicle
        state = 2;   // move to state 2(parking)
        park_state = 0;
        cnt = 0;
    }

}
```

Fig 9. lane tracer

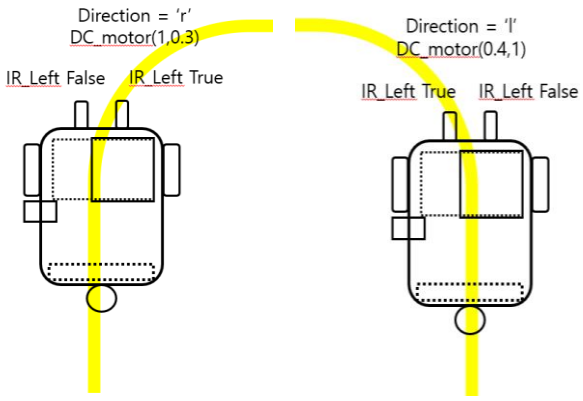


Fig 10. rotation interrupt

3. Parking logic

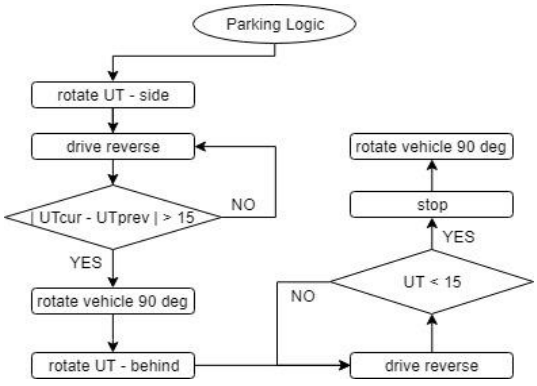


Fig 11. Parking Flowchart

After the vehicle meets the wall, the state goes to parking mode. The parking logic is shown as flowchart at Fig 11.
As soon as the state goes to parking mode, the ultrasonic sensor rotates 90 degrees to face the side (Fig 12).
Then vehicle goes backwards and compares the current and past UT sensor values to find the edge (Fig 13).
When edge is detected, the vehicle rotates 90 degrees to prepare to back into the parking space (Fig 14).
Then, vehicle goes backwards until it is completely inside the parking space (Fig 15).
Finally, the vehicle rotates 90 degrees so that it is parked parallel to the wall (Fig 16). The code is attached in the appendix.

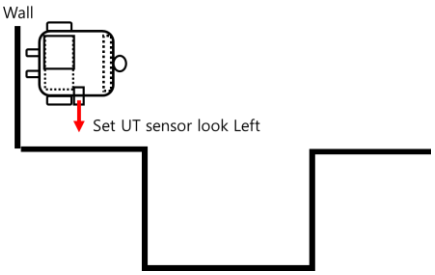


Fig 12. Parking step 1

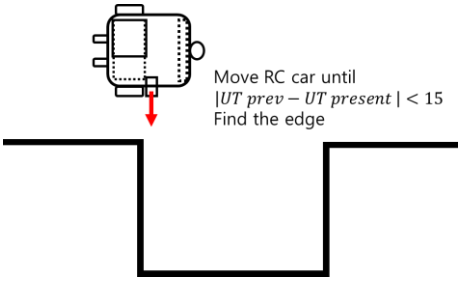


Fig 13. Parking step 2

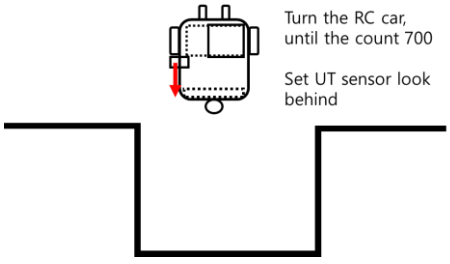


Fig 14. Parking step 3

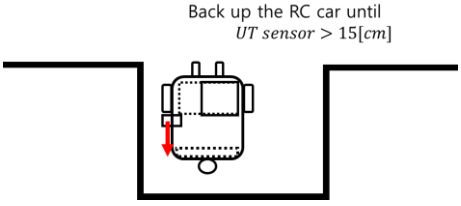


Fig 15. Parking step 4

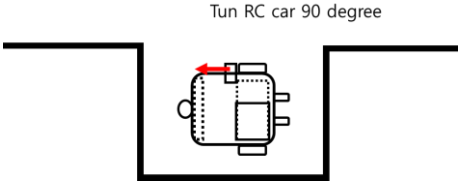


Fig 16. Parking step 5

IV. Results & Analysis

The line tracing RC car showed prominent result, since it didn't have any problem and came in as second place. Also, since we only used 3 sensors, it can be said that our model is efficient. One reason of the result was that we had checked all the sensors and motors working at the same time before start assembling the vehicle. Other teams have complained that adding sensors to fully-assembled vehicle was frustrating. Another reason is that we focused on efficiency and stability when building the algorithm. The crucial tasks such as lane tracing was set as number one prior so that it does not wobble during line tracing. This saved a lot of time and made the vehicle able to drive smoothly.

Table 3. Evaluation result

evaluation	result
Bluetooth start	O
Track completion time	30 sec (2nd rank)
Line departure	None
Contact while parking	None

Though the result came out well, there were many troubles we had to overcome. First, we had trouble in setting delay. In parking situations, we first used delay function between steps. This brought a critical problem, since it gets other timers tangled. To fix this problem, I declared a global variable that timer 9 increases it by one every 1 millisecond. I used this variable to replace delay function, and it brought much more consistent and reliable result.

Second, there was a problem rotating differently every time we park the vehicle. This was due to heavy rear part, so we have moved the battery to the front part of the vehicle. This result in weight distributing mostly between the wheels, which means that the mass moment of inertia is decreased. After moving the battery, the vehicle was able to rotate with low rotational force, and caused more consistent result.

V. Conclusion

In this project, we have designed a line tracing and auto-parking RC car. We were able to design stable and fast vehicle, and finish the track in 2nd. Our plan to open all the sensors and test them all at the same time before moving on to assembly has saved us a lot of time. Also, using one ultrasonic sensor has reduced the number of peripherals and waste of power on multiple ultrasonic sensors. We have concluded that our tightly set plan made our result more prominent.

From this project we were able to fully train ourselves to implement multi-sensor, multi-interrupt using application on embedded system. We have concluded that most of the electronic control we have learned in mechatronics can be implemented on embedded system as well, such as multi-loop PID controller.

In future, we are looking forward to use embedded board and FPGA together to design embedded system that can implement deep learning based algorithm.

Appendix

```
void park_vehicle(void)
{
    switch(park_state)
    {
        case 0: // rotate UT
            if(cnt < 1200 && cnt > 500)
            {
                PWM_pulsewidth_ms(&pwm_RC, RC_left);
                UT_prev = UT_distance;
                park_state = 1;
            }
            break;

        case 1: // find edge
            if( abs(UT_distance - UT_prev) < 15)
            {
                UT_prev = UT_distance;
                drive('b');
                cnt = 0;
            }
            else if(cnt > 315)
            {
                park_state = 2;
                cnt = 0;
            }
            break;
        case 2: // turn 90 degrees
                if(cnt < 700)
                {
                    DC_motor(0,-0.8);
                }
                else if(cnt < 1300)
                {
                    drive('x');
                    PWM_pulsewidth_ms(&pwm_RC, RC_behind);
                }
                else
                {
                    park_state = 3;
                    cnt = 0;
                }
                break;

        case 3: // go inside the parking lot
            if(UT_distance > 15)
            {
                drive('b');
            }
            else
            {
                drive('x');
                park_state = 4;
                cnt = 0;
            }
            break;
        case 4: // 90 degree again
            if(cnt < 800)
                DC_motor(0.8,-0.8);
            else
            {
                drive('x');
                state = 0;
                park_state = 0;
            }
            break;
    }
}
```

Fig 17. Park Vehicle