



第四届  
全国大学生集成电路创新创业大赛  
CICIEC

项目设计报告

参赛题目: RISC-V 挑战杯子赛题一

队伍编号: AGD103942

团队名称: light chaser

联系人: 林沐晗

电 话: \_\_\_\_\_

邮 箱: muhanlin@link.cuhk.edu.cn

## 目录

项目总述.....	5
软硬件说明.....	5
英文说明.....	5
1. 运动控制.....	6
1.1 运动实现.....	6
1.1.1 通过输出高低电平控制马达转动.....	6
1.1.2 通过 PWM 控制小车转速.....	6
1.2 运动距离和角度控制.....	9
1.2.1 小车的速度计算.....	9
2. 循迹控制.....	10
2.1 循迹方法流程图.....	10
2.2 模块使用.....	10
2.2.1 红外模块.....	10
2.3 核心控制代码说明.....	11
2.3.1 小车控制主程序.....	11
2.3.2 小车 PWM 函数的说明.....	13
2.4 使用方法.....	15
3. RTOS 的使用.....	16
3.1 FreeRTOS 介绍.....	16
3.2 该项目应用 FreeRTOS 解决的问题.....	18
3.3 FreeRTOS 移植说明.....	18
3.2 多线程与多任务的实现.....	20
3.2.1 多线程与多任务功能说明.....	20
3.2.2 现象.....	20
3.2.3 实现方法.....	20
3.3 多任务、队列与通信.....	21
3.3.1 功能说明.....	21
3.3.2 目标现象.....	21

3.3.2 实现方法.....	22
3.4 多任务、队列与通信的原理说明.....	23
3.4.1 多任务实现的原理说明.....	23
3.4.2 队列与通信的原理说明.....	24
4. 总结与展望.....	24
4.1 总结.....	24
4.2 展望.....	24

## 图标目录

图 1: 小车马达.....	6
图 2: 电机 GPIO 说明.....	7
图 3: 金手指引脚说明.....	8
图 4: 小车主轮缺失 PWM.....	8
图 5: 小车速度随 PWM 变化.....	9
图 6: 程序设计流程图.....	10
图 7: 红外传感器.....	11
图 8: FreeRTOS 移植.....	19
图 9: 移植成功界面.....	20
图 10: FreeRTOS 循迹执行界面.....	20
图 11: 程序执行界面.....	22

## 项目总述

RISC-V 开源架构为物联网时代提供了新的芯片选择，近年来受到越来越多的关注。本项目通过使用 RISC-V SiFive Learn Inventor 芯片，首先通过对寄存器控制，使用红外传感器实现了小车循线行驶。在加入 FreeRTOS 实时多线程操作系统后，实现了芯片与小车的多线程通信，达到多任务并行的效果。本文对该项目实施流程进行了详细介绍，并附图展示效果。

## 软硬件说明

开发板	Sifive Learn Inventor
集成开发环境	Freedom Studio v10.3.1 Linux
小车	柴火 BitCar microbit 智能小车套件
电脑操作系统	Ubuntu 20.04 LTS

附注：项目全部主程序位于 *src/exmaple-freertos-minimal/exmaple-freertos-minimal.c* 文件中

## 英文说明

为方便阅读，本文中英文函数和程序包、程序文件名使用 **加粗斜体** 表示，缩写英文名称使用 **加粗** 表示，其他英文路径、名称等使用 *斜体* 表示。

# 1. 运动控制

## 1.1 运动实现

### 1.1.1 通过输出高低电平控制马达转动

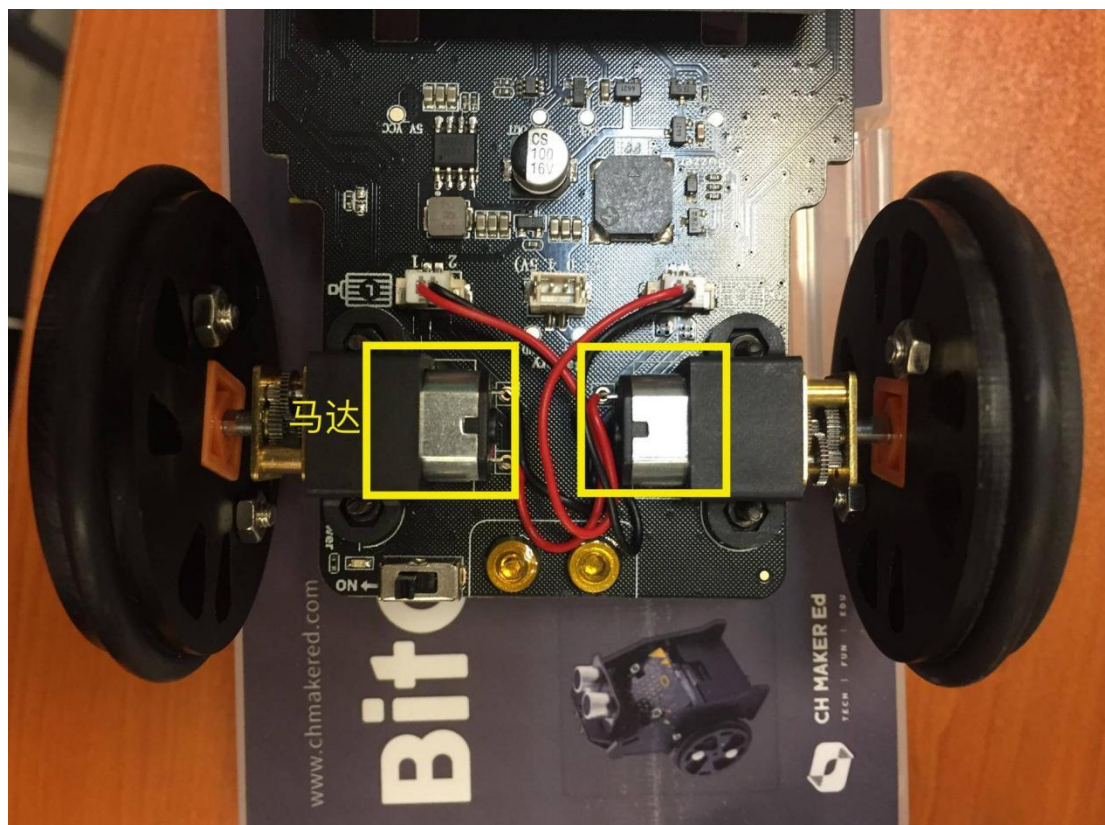


图 1：小车马达

小车左右轮的前转与后转分别由不同的 GPIO 口控制，共对应 4 个 GPIO 引脚。按照官方文档进行地址计算，得到控制四个引脚的几个寄存器地址。通过将 1 写入寄存器相关位置使能四个 GPIO 口。然后将 1 写入存储 GPIO 口输出值的寄存器，便可使 GPIO 口输出高电平，从而控制相应的轮子转动；若是写入 0，则可使电机停止转动。

### 1.1.2 通过 PWM 控制小车转速

鉴于开发板本身 PWM 不完整，本队在程序中自主设计了 PWM 函数。函数输入量 *speed* 的大小决定了 PWM 的占空比的大小。占空比是指在一个脉冲循环内，通电时间相对于总时间所占的比例。占空比越大，则小车电机转动速度越

快。本程序设定 *speed* 最大值为 255，最小值为 0。当输入 *speed* 为 0 时，小车马达全速运转，当输入 *speed* 为 255 时，小车马达不运转。下图一为芯片各个 GPIO 口连出的 PWM 接口，红色部分为小车电机对应的 GPIO 口，缺失 PWM 的 GPIO 口为 4 和 5。图二为芯片 GPIO 口对应的小车的 *pin* 口，4、5 对应的是 *p13*、*p14*，图三为开发板上 *pin* 口，可见 *p13*、*p14* 对应小车左轮，小车左轮缺失 PWM。因此，本队自主设计了四个 PWM 函数来实现小车直行，左转，右转，倒车的控制，详细原理将在核心代码控制部分讲解。

Copyright © 2019, SiFive Inc. All rights reserved.

GPIO Number	IOF0	IOF1
0		PWM0_PWM0
1		PWM0_PWM1
2	SPI1_CS0	PWM0_PWM2
3	SPI1_DQ0	PWM0_PWM3
4	SPI1_DQ1	
5	SPI1_SCK	
6	SPI1_DQ2	
7	SPI1_DQ3	
8	SPI1_CS1	
9	SPI1_CS2	
10	SPI1_CS3	PWM2_PWM0
11		PWM2_PWM1
12	I2C0_SDA	PWM2_PWM2
13	I2C0_SCL	PWM2_PWM3
14		
15		
16	UART0_RX	
17	UART0_TX	
18	UART1_TX	
19		PWM1_PWM1
20		PWM1_PWM0
21		PWM1_PWM2
22		PWM1_PWM3
23	UART1_RX	
24		
25		
26	SPI2_CS0	
27	SPI2_DQ0	
28	SPI2_DQ1	
29	SPI2_SCK	
30	SPI2_DQ2	
31	SPI2_DQ3	

Table 53: GPIO IOF Mapping

图 2：电机 GPIO 说明





## 1.2 运动距离和角度控制

### 1.2.1 小车的速度计算

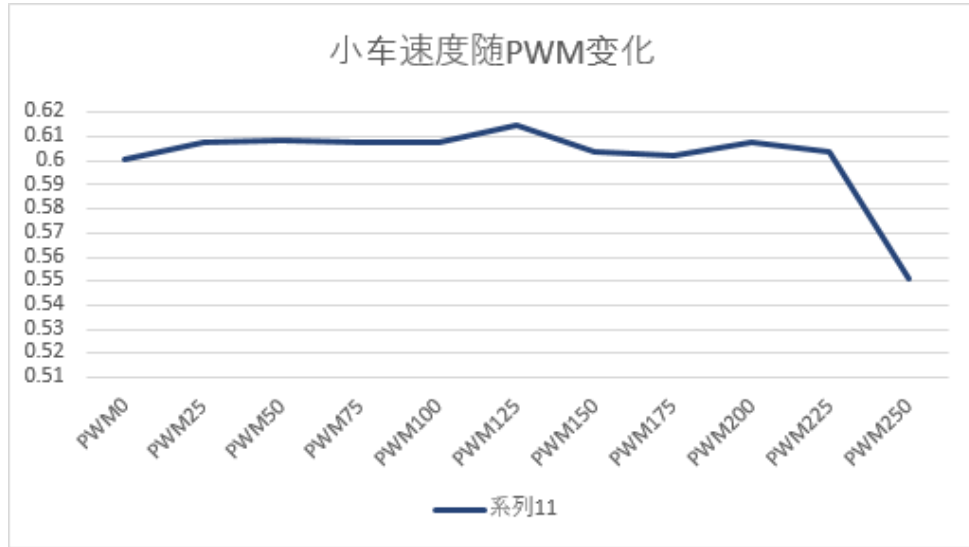


图 5：小车速度随 PWM 变化

本队做了若干实验，采用不同大小的 PWM 计算小车的运动速度。小车的速度基本在  $0.6m/s$  和  $0.61m/s$  之间。使用 `PWM_run (int speed)` 函数控制小车以一定速度直行，再使用延时函数 `wait_ms(uint64_t ms)` 控制小车以该速度直行的时间，则可控制小车的行驶距离。对于 `PWM_run (int speed)` 和 `PWM_back (int speed)` 函数工作原理的讲解见下文核心代码讲解部分之 2.3.2 小车 PWM 函数的说明。

### 1.2.2 小车的运动角度计算

本队在转弯时使用的是只转动一个轮子，所以小车运动角度可以使用小车的运动线速度计算得出： $v = \omega r$ 。小车的半径  $r$  为  $3.675cm$ ，取小车基本运行速度为  $0.61m/s$ ，则小车运动角速度  $\omega$  为  $16.5986rad/s$ 。使用 `PWM_left (int speed)` 或 `PWM_right (int speed)` 函数控制小车以一定角速度转弯，再使用延时函数 `wait_ms(uint64_t ms)` 控制小车以该角速度运行的时间，则可控制小车的转动角度。对于 `PWM_left (int speed)` 和 `PWM_right (int speed)` 函数工作原理的讲解见下文核心代码讲解部分之 2.3.2 小车 PWM 函数的说明。

## 2. 循迹控制

### 2.1 循迹方法流程图

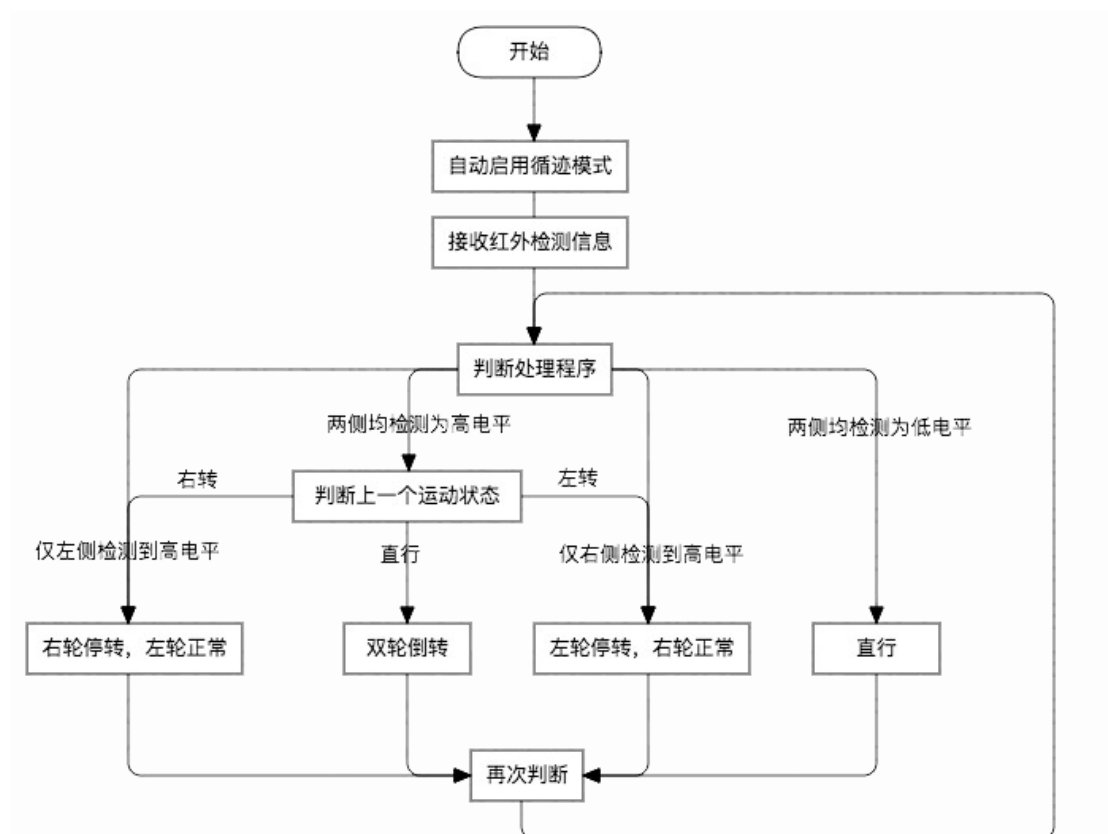


图 6: 程序设计流程图

### 2.2 模块使用

#### 2.2.1 红外模块

$P1$ 、 $P2$  两个  $Pin$  口一端分别连接左右两个红外传感器，另一端连接两个  $gpio$  引脚。当传感器在黑色轨道上时，传感器将接收到弱红外线信号，则对应的  $gpio$  引脚输出低电平，寄存器相关值显示为 0。当传感器不在黑色轨道上时，传感器将接收到强红外线信号，则对应的  $gpio$  引脚输出高电平，寄存器相关值显示为 1。程序计算地址，获得存储  $gpio$  引脚输入值的寄存器位的对应地址，再读取

该地址内的值，来判断小车运动状态，从而控制小车做出不同的反应，实现循迹功能。

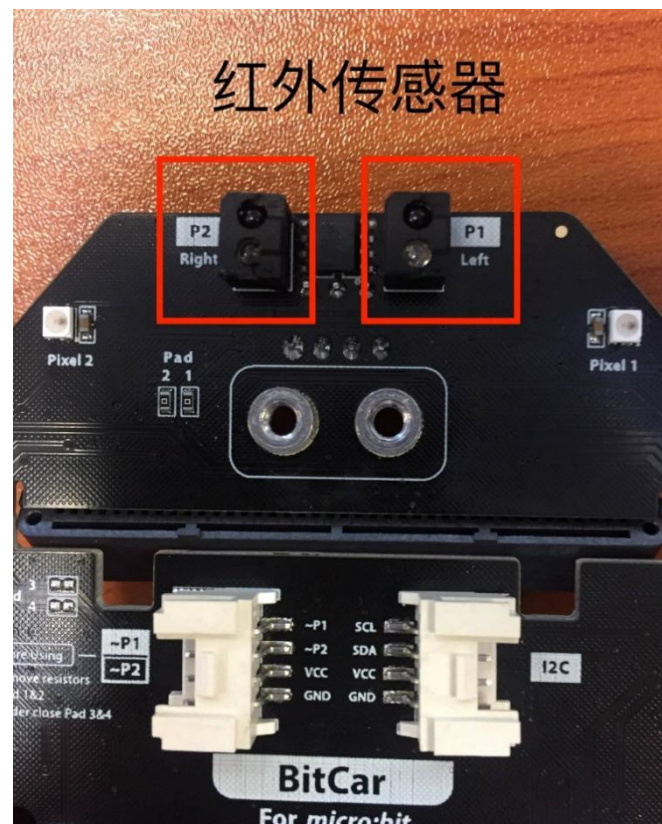


图 7：红外传感器

## 2.3 核心控制代码说明

### 2.3.1 小车控制主程序

```
1. void tracking_car_control()
2. {
3.     uint8_t msg;
4.     int state;
5.     if(Message_Queue!=NULL)
6.     {
7.         write(STDOUT_FILENO, "Queue exists!", strlen("Queue exists!")
8.         );//队列创建成功
9.         if(xQueueReceive(Message_Queue, &msg, portMAX_DELAY))//接受队
10.        列里的信息
11.        {
12.            PWM_run(250); //以 0.61m/s 的速度直行
```

```

13.                 state = 1;
14.             }
15.             if(msg==10)//小车左轮出轨道
16.             {
17.                 state = 2;
18.                 PWM_left(250);//左轮速度约为 0.61m/s
19.                 GPIO_SET(car_motor_right_back
0, output);
20.                 GPIO_SET(car_motor_right_forward
0, o
utput);//右轮停转
21.             }
22.             if(msg==01)//小车右轮出轨道
23.             {
24.                 state = 3;
25.                 PWM_right(250);//右轮速度约为 0.61m/s
26.                 GPIO_SET(car_motor_left_back
, 0, output);
27.                 GPIO_SET(car_motor_left_forward
0,
output);//左轮停转
28.             }
29.             if((msg==11)&&(state == 1))//小车双轮出轨道且前一步直行
30.             {
31.                 PWM_back(250);//以 0.61m/s 的速度倒车
32.             }
33.         }
34.     }
35.     else{
36.         write(STDOUT_FILENO, "No queue!", strlen("No queue!"));//队
列创建失败
37.     }
38. }

```

打开电源开关后，小车自动开启循迹程序。在循迹程序中，每一个循环内，小车先接收队列中传输的内容，得到 *msg* 的值，即红外线检测结果。然后小车根据红外线探测结果来控制电机下一步输出。当两侧红外传感器都检测到低电平时，*msg* 的值为 00，小车行驶在黑线即赛道上，程序控制小车直行，变量 *state* 被标记为 1。当小车仅左边传感器检测到高电平时，*msg* 的值为 10，小车左侧驶出赛道，程序控制左轮正常运转、右轮停转，小车右转，变量 *state* 被标记为 2；当小车仅右边传感器检测到高电平时，*msg*

的值为 01，小车右侧驶出赛道，程序控制小车右轮正常运转、左轮停转，小车主转，变量 *state* 被标记为 3；当小车两边传感器均检测到高电平、*msg* 的值为 11 时，则判断上一步小车运动状态，即读取 *state* 的当前值。若 *state* 指明上一步左转则继续左转，上一步右转则继续右转，上一步直行则倒车。

### 2.3.2 小车 PWM 函数的说明

```

1. void PWM_run(int speed)//控制小车直行
2. {
3.     for(int i = 0;i < 255;i++)
4.         if(i > speed)//speed 为低电平输出的时长
5.         {
6.             GPIO_SET(car_motor_left_back , 0, output);
7.             GPIO_SET(car_motor_left_forward , 1, output);//小车主
            轮前进
8.             GPIO_SET(car_motor_right_back , 0, output);
9.             GPIO_SET(car_motor_right_forward , 1, output);//小车主
            轮前进
10.
11.         }
12.     else
13.     {
14.         GPIO_SET(car_motor_left_back , 0, output);
15.         GPIO_SET(car_motor_left_forward , 0, output);//小车主轮
            停止
16.         GPIO_SET(car_motor_right_back , 0, output);
17.         GPIO_SET(car_motor_right_forward , 0, output);//小车主
            轮停止
18.
19.     }
20. }
21. void PWM_right(int speed)//控制小车主转
22. {
23.     for(int i = 0;i < 255;i++)
24.         if(i > speed)//speed 为低电平输出的时长
25.         {
26.             GPIO_SET(car_motor_right_back , 0, output);
27.             GPIO_SET(car_motor_right_forward , 1, output);//小车主
            轮前转
28.         }
29.     else

```

```

30.         {
31.             GPIO_SET(car_motor_right_back      , 0, output);
32.             GPIO_SET(car_motor_right_forward    , 0, output); //小车右
    轮停转
33.         }
34. }
35. void PWM_left(int speed) //控制小车右转
36. {
37.     for(int i = 0; i < 255; i++)
38.         if(i > speed) //speed 为低电平输出的时长
39.         {
40.             GPIO_SET(car_motor_left_back        , 0, output);
41.             GPIO_SET(car_motor_left_forward      , 1, output); //小车主
    轮前转
42.         }
43.         else
44.         {
45.             GPIO_SET(car_motor_left_back        , 0, output);
46.             GPIO_SET(car_motor_left_forward      , 0, output); //小车主
    轮停转
47.         }
48. }
49. void PWM_back(int speed) //控制小车后退
50. {
51.     for(int i = 0; i < 255; i++)
52.         if(i > speed) //speed 为低电平输出的时长
53.         {
54.             GPIO_SET(car_motor_left_back        , 1, output);
55.             GPIO_SET(car_motor_left_forward      , 0, output); //小车主
    轮后退
56.             GPIO_SET(car_motor_right_back       , 1, output);
57.             GPIO_SET(car_motor_right_forward     , 0, output); //小车主
    轮后退
58.         }
59.         else
60.         {
61.             GPIO_SET(car_motor_left_back        , 0, output);
62.             GPIO_SET(car_motor_left_forward      , 0, output); //小车主
    轮停止
63.             GPIO_SET(car_motor_right_back       , 0, output);
64.             GPIO_SET(car_motor_right_forward     , 0, output); //小车主
    轮停止
65.         }
66. }

```

- *PWM\_run*: 控制小车以一定速度直行运动。
- *PWM\_left*: 控制小车以一定速度右转：左轮以一定速度运转，右轮不运转。
- *PWM\_right*: 控制小车以一定速度左转：右轮以一定速度运转，左轮不运转。
- *PWM\_back*: 控制小车以一定速度倒车。
- 函数输入值 *speed* : 控制低电平输出时间。
- *GPIO\_SET (uint32\_t pin\_num, uint32\_t pin\_val, uint32\_t pin\_model)*:

使能相应 **GPIO** 引脚，并设置其为输入或输出以及其输入或输出的值（均通过计算得到所需寄存器地址后将 1 或 0 写入寄存器相应位实现）

1. 参数 *pin\_num*: 设置的 **GPIO** 引脚的序号。*car\_motor\_left\_forward*、*car\_motor\_left\_backward*、*car\_motor\_right\_forward*、*car\_motor\_right\_backward* 均为存储电机对应的 **GPIO** 引脚序号的常量。
  2. 参数 *pin\_val*: 该引脚的输出/输入值。1 表示高电平，0 表示低电平。
  3. 参数 *pin\_model*: 该引脚为输入或输出。可填入的值为 *input* 和 *output*（在相关文件中已将 *output* 定义为 1，*input* 定义为 2）。
- **PWM 函数控制原理**：在 **PWM** 函数中，只有当 *i* 大于 *speed* 时，驱动电机的 **GPIO** 引脚才会输出高电压。因此 *speed* 越大，输出高电压的时间越短。通过使用 *for* 循环，能够模拟一定占空比的交变电流，从而实现 **PWM** 控制小车速度。

## 2.4 使用方法

- 程序烧录

(1) 在 **Freedom Studio** 中以导入已存在的文件夹形式导入小车文件，在 *run configuration* 处配置 *JLink* 的相关设置。

(2) 连接开发板至电脑。

(3) 点击运行按钮，代码自动编译并烧录进开发板中

- 场地布置与小车的组装

打开柴火 **BitCar microbit** 智能小车套件的轨道或自主用黑胶布在浅色地板上粘贴出轨道的形状。（注：对于宽度合适的轨道，轨道规模（直道长度、弯道半径与弧度）不同，小车巡线最佳速度也不同。）

小车的组装参照柴火 **BitCar microbit** 智能小车套件的小车安装说明书。小车拥有两种供电方式，其中用电池供电比较便捷，但是会有电压不稳或不足以给开发板供电导致程序失灵的情况，使用外置电源如充电宝就可以避免上述情况。

- 运行

打开开关，将小车红外传感器对准黑线，使红外传感器感应灯熄灭，即可自动实现循迹。

## 3. RTOS 的使用

### 3.1 FreeRTOS 介绍

**FreeRTOS** 是一个能满足嵌入式硬实时要求的实时内核（或调度器）。它实现了应用的多线程设计要求。对于仅有一个内核的处理器来说，任意时间只有一个线程能被执行。内核根据设计者分配给各个任务的优先级高低来决定哪个线程先被执行。任务优先级能保证一个应用满足其执行时的时限要求。**FreeRTOS** 有如下几个优势：

- (1) **FreeRTOS** 提供 **API** 函数来完成计时任务。这可以使应用程序更加简洁，减少应用程序导致的错误，增强软件的可控性。
- (2) 每个线程都是独立的，都有其明确定义的任务。
- (3) 线程之间有定义完善的交互界面，使得团队开发更加简单。
- (4) 当任务为定义完善且具有简洁的交互界面时，它们能够被独立测试。



(5) 源码被很好地模块化，具有较少的依赖性，这使得开发者能够更轻松地将这些代码重复使用。

(6) 使用一个内核允许软件完全由事件驱动，所以没有处理时间会被浪费在投票决定执行事件上。只有当某些必要条件出现，代码才会被执行。同时，**FreeRTOS** 提供定时器时间中断以及切换任务的功能。

(7) 当任务调度器开启，空闲任务会被自动创建。每当没有应用任务需要被执行时，空闲任务便会执行。空闲任务可以被用于计算空闲时间，执行背景检查，或让处理器进入低功耗模式。

(8) 使用 **FreeRTOS** 提高效率能使处理器更多时候处于低功耗模式下。这能使处理器的能源消耗显著降低。**FreeRTOS** 还提供了特殊的 *tickless* 模式。该模式下，处理器进入的低功耗模式消耗的能量要低于其他任何时候，并且处理器能够在该模式下停留更长时间。

(9) 简单的设计模式允许在一个应用实现周期性的、持续的、事件驱动的混合处理方式。另外，通过选择合适的任务和中断优先级能够满足软实时或硬实时要求。

**FreeRTOS** 还有如下特点：

- (1) 抢占式或协程式操作
- (2) 灵活的任务优先级分配
- (3) 灵活、迅速、轻量级的任务通知机制
- (4) 使用队列
- (5) 应用二值信号量、计数信号量、互斥信号量与递归互斥信号量
- (6) 软件定时器
- (7) 事件组
- (8) 任务钩子函数与空闲任务钩子函数
- (9) 堆栈溢出检查
- (10) 跟踪记录与任务运行时间统计
- (11) 完备的中断设计
- (12) 为极低功耗应用设计的 *tickless* 模式

### 3.2 该项目应用 FreeRTOS 解决的问题

仅使用小车裸机程序时，小车无法同时执行多项任务。这会导致小车执行任务的实时性和效率较低，浪费了处理时间和内存资源，能耗较高。同时，对于多项任务的调度工作完全依赖于开发者编写的程序，不仅增加了开发者的工作负担，也使得任务程序更繁杂，更容易产生错误。

本项目应用实时操作系统很好地解决了这些问题。当小车执行延时程序、电机持续输出相同电压时，处理器实际上是处于空闲状态。应用 RTOS 可以将该时段利用起来：延时的任务进入阻塞态，处理器另从就绪列表中选择优先级最高的任务来执行。这样就实现了“多任务同时进行”的程序执行效果，节省处理时间。FreeRTOS 的任务调度器可以依据优先级协调各任务的执行，并且 FreeRTOS 提供计时服务，这使得应用程序更简洁，软件表现更稳定，大大降低开发者负担。同时，RTOS 为各项任务提供合理的内存管理服务，使得内存利用更高效、更具可控性。

该项目对于 FreeRTOS 的应用开发分为三个阶段：移植，多任务创建与执行，队列创建与通信。接下来本文将对这三个阶段进行详细说明。

### 3.3 FreeRTOS 移植说明

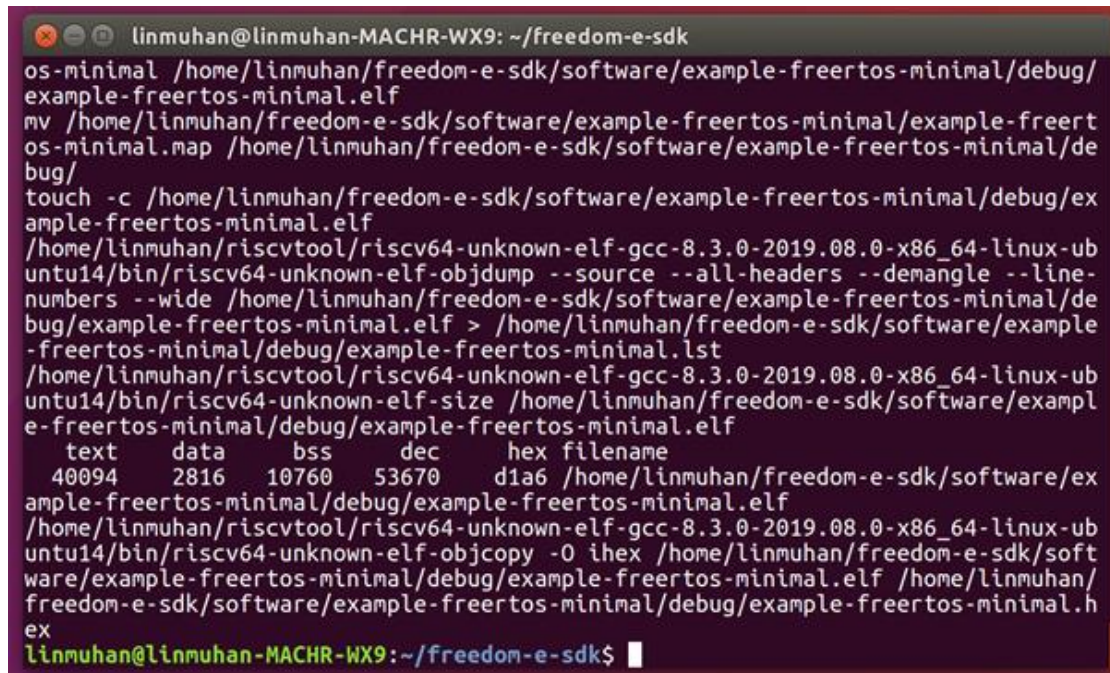
(1) 下载 *freedom-e-sdk* 程序包，在 ubuntu 终端设置编译所需路径：

```
1. export RISCVC_PATH=~/.riscvtool/riscv-openocd-0.10.0-2019.08.2-x86_64-linux-ubuntu14
2. export PATH=$PATH:~/.riscvtool/riscv-qemu-4.1.0-2019.08.0-x86_64-linux-ubuntu14/bin
3. export RISCVC_PATH=~/.riscvtool/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14/
```

(2) 在 ubuntu 终端使用如下命令编译 SiFive learn inventor 对应的 FreeRTOS 示例程序 *example-freertos-minimal*：

```
1. make PROGRAM=example-freertos-minimal TARGET=sifive-hifive1-revb LINK_TARGET=freertos software
```

编译得到一个 *elf* 文件，该文件自动保存在该示例程序文件夹下的 *debug* 文件夹内。



```
linmuhan@linmuhan-MACHR-WX9: ~/freedom-e-sdk
os-minimal /home/linmuhan/freedom-e-sdk/software/example-freertos-minimal/debug/
example-freertos-minimal.elf
mv /home/linmuhan/freedom-e-sdk/software/example-freertos-minimal/example-freert
os-minimal.map /home/linmuhan/freedom-e-sdk/software/example-freertos-minimal/de
bug/
touch -c /home/linmuhan/freedom-e-sdk/software/example-freertos-minimal/debug/ex
ample-freertos-minimal.elf
/home/linmuhan/riscvtool/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ub
untu14/bin/riscv64-unknown-elf-objdump --source --all-headers --demangle --line
numbers --wide /home/linmuhan/freedom-e-sdk/software/example-freertos-minimal/de
bug/example-freertos-minimal.elf > /home/linmuhan/freedom-e-sdk/software/example
-freertos-minimal/debug/example-freertos-minimal.lst
/home/linmuhan/riscvtool/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ub
untu14/bin/riscv64-unknown-elf-size /home/linmuhan/freedom-e-sdk/software/exampl
e-freertos-minimal/debug/example-freertos-minimal.elf
      text    data    bss     dec     hex filename
  40094      2816   10760   53670   dia6 /home/linmuhan/freedom-e-sdk/software/ex
ample-freertos-minimal/debug/example-freertos-minimal.elf
/home/linmuhan/riscvtool/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ub
untu14/bin/riscv64-unknown-elf-objcopy -O ihex /home/linmuhan/freedom-e-sdk/soft
ware/example-freertos-minimal/debug/example-freertos-minimal.elf /home/linmuhan/
freedom-e-sdk/software/example-freertos-minimal/debug/example-freertos-minimal.h
ex
linmuhan@linmuhan-MACHR-WX9:~/freedom-e-sdk$
```

图 8: FreeRTOS 移植

(3) 打开 FreedomStudio，以 “C/C++ Executable” 的形式导入 *elf* 文件，生成一个单独的工程。将该工程烧录到开发板上，串口打印如下信息，则说明 FreeRTOS 移植成功。

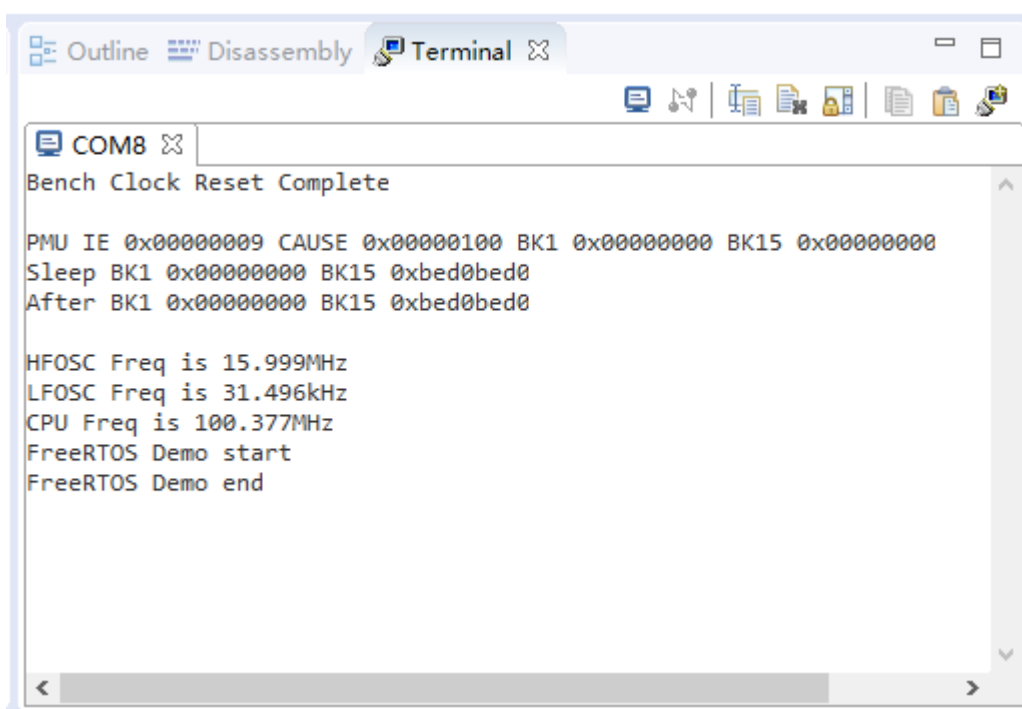


图 9：移植成功界面

## 3.2 多线程与多任务的实现

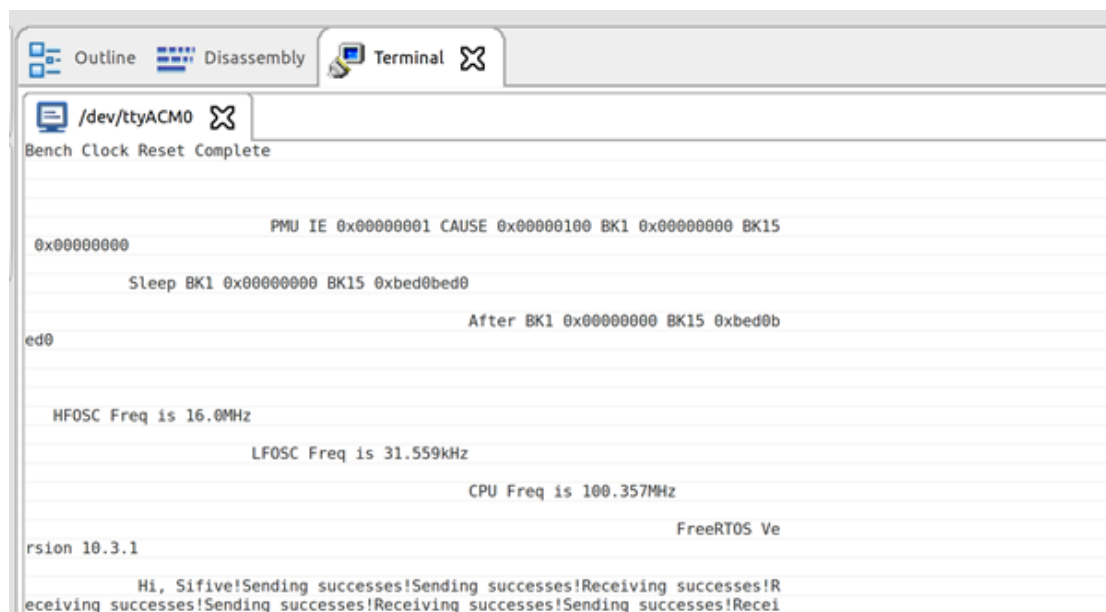
### 3.2.1 多线程与多任务功能说明

该项目中创建了三个线程，线程 *Print\_task* 在程序开始时打印 RTOS 版本信息以及 “Hi SiFive” 字样，线程 *Infrared\_task* 完成红外线实时探测、信息发送任务，线程 *Car\_task* 接收红外信息并控制小车电机转动。

### 3.2.2 现象

串口先打印出 RTOS 版本信息以及 “Hi SiFive” 字样，接着小车根据红外线传感器探测结果进行相应运动，完成巡线任务。

- 现象截图



```
Outline Disassembly Terminal X
/dev/ttyACM0 X
Bench Clock Reset Complete

PMU IE 0x00000001 CAUSE 0x00000100 BK1 0x00000000 BK15
0x00000000
Sleep BK1 0x00000000 BK15 0xbcd0bcd0
After BK1 0x00000000 BK15 0xbcd0b
ed0

HFOSC Freq is 16.0MHz
LFOSC Freq is 31.559kHz
CPU Freq is 100.357MHz
FreeRTOS Ve
rsion 10.3.1
Hi, Sifive!Sending successes!Sending successes!Receiving successes!R
eceiving successes!Sending successes!Receiving successes!Sending successes!Recei
```

图 10：FreeRTOS 循迹执行界面

### 3.2.3 实现方法

(1) 将目标任务相关代码及库文件移植到示例程序包 *example-freertos-minimal* 中：目标任务主函数直接复制粘贴到示例程序主函数文件中。

根据引用路径将所需库文件放入示例程序包相应位置。在 `ubuntu` 终端单独编译所需库文件得到 `<FILE>.o` 文件，接着使用 `ar` 命令将所得 `<FILE>.o` 文件并入 `freedom-e-sdk/bsp/sifive-hifive1-revb/install/lib/debug` 中的 `libmetal.a` 文件中。

(2) 在 `main` 函数中多次调用 `xTaskCreate` 函数建立多个任务（或建立单个任务 `start_task`，再在该任务中创建其他任务），给每个任务分配合适大小的堆栈以及合适的优先级，保证任务能成功创建并正常运行。可在任务末尾调用 `uxTaskGetStackHighWaterMark` 来估计该任务所需堆栈空间，进而对分配给该任务的堆栈大小进行调整。

对于需要几乎同时运行的任务，应在任务函数末尾调用 `vTaskDelay` 函数设置合适的任务阻塞时间，使得 CPU 能够在这段时间内完成需要并行的任务。

(3) 完成任务创建后，在 `main` 函数中调用 `vTaskStartScheduler` 函数开启任务调度器。

### 3.3 多任务、队列与通信

#### 3.3.1 功能说明

该项目程序中建立了一个消息队列，用于传感器与小车电机间的消息传递。

#### 3.3.2 目标现象

小车能正常巡线。

目标实现截图：

- 发送信息代码

```
• result=read_pin_val();
• err=xQueueSend(Message_Queue, &result, 0);
• if(err==errQUEUE_FULL)
• {
•     write(STDOUT_FILENO, "Queue is full. Transmitting fails!", strlen("Queue
    is full. Transmitting fails!"));
• }
• else
• {
•     write(STDOUT_FILENO, "Sending successes!", strlen("Sending successes!"));
```

```

• }
• • 接收信息代码
• if(Message_Queue!=NULL)
• {
•     if(xQueueReceive(Message_Queue,&msg,portMAX_DELAY))
•     {
•         write(STDOUT_FILENO,"Queue exists!",strlen("Queue
exists!"))

```

- 程序运行输出

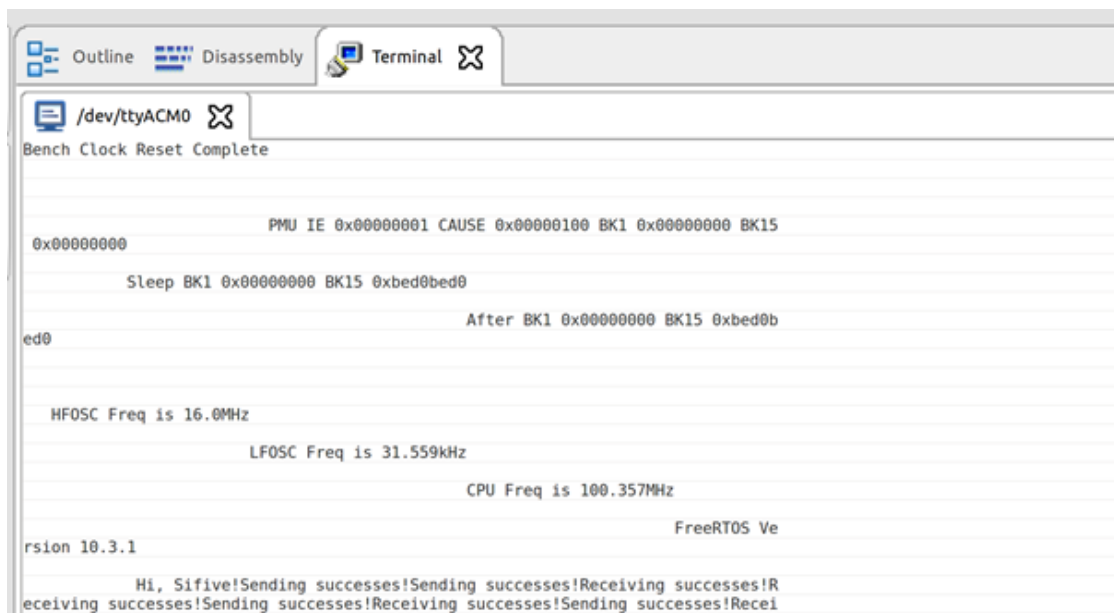


图 11：程序执行界面

### 3.3.2 实现方法

(1) 在 *main* 函数或是 *start\_task* 中调用 *xQueueCreate* 函数创建队列，返回指向队列的队列句柄。设置合适的队列长度以及队列中的消息长度。

(2) 在红外线探测任务中，每探测一次就调用 *xQueueSend* 函数，将记录探测结果的局部变量地址传入消息队列。为了保证探测的实时性，传入阻塞时间设置为 0。

(3) 巡线任务中，程序先调用 *xQueueReceive* 函数，从队列中读取探测结果存储地址，从而获得探测结果，再根据结果控制小车进行下一步运动。出队阻塞时间设置为最大，从而使得小车严格按照红外探测结果运动。

### 3.4 多任务、队列与通信的原理说明

#### 3.4.1 多任务实现的原理说明

事实上，在任意时间 CPU 总是仅执行一个任务的程序。RTOS 可能在某个任务循环中阻塞或挂起该任务，然后去执行另一任务，从而实现多任务同时进行的效应。任务调度由任务调度器函数来实现。

任务有四种状态：运行态、就绪态、阻塞态和挂起态。每个任务都拥有指向自己的状态列表项和事件列表项。这些列表项从属于某个列表，例如就绪列表、等待某个事件列表等等。它们记录了各个任务的状态或是阻塞状态下等待的事件。任务调度器总是会从就绪列表选择优先级最高的任务来执行。当某一事件发生，等待该事件的任务 *A* 将会从阻塞态进入就绪态，实质是其状态列表项从一个列表中删除后进入另一个列表。如果任务 *A* 的优先级高于当前执行的任务 *B* 的优先级，且抢占式内核对应的宏被设为 1, *A* 将会夺取 *B* 的执行权转入运行态，任务 *B* 的执行因此被中断。若是任务函数中调用了任务挂起函数，则目标任务将被挂起，CPU 使用权亦将被转让。只有当其他任务调用函数来恢复这个被挂起的任务时，该任务会重新进入就绪态等待被执行。

对于同一优先级的任务调度，有两种情况。若是使用时间片管理（相关宏设为 1），则 CPU 在每个时间片执行一个任务，该优先级的任务轮流被执行。若是不使用时间片管理（相关宏设为 0），则任务转换仅仅发生在更高优先级任务抢占或当前任务进入阻塞或挂起态之时。

在这个项目中，首先被执行的是开始任务 *Start\_Task*，优先级为 1。开始任务首先创建出队列和优先级为 5 的 *Print\_task*，打印所需字符串。

*Print\_task* 执行完成，该任务被删除，CPU 使用权转交给 *Start\_task*。接着优先级为 3 的红外检测任务被创建，CPU 使用权被转交给红外检测任务。红外检测任务的首个循环完成后，*vTaskDelay* 函数被调用，该任务进入阻塞态。此时 CPU 创建 *Car\_task*，优先级为 4，高于红外探测任务，优先被 CPU 运行。*Car\_task* 也在第一个循环末尾进入延时阻塞状态。在 *Car\_task* 的延时段中，优先级低于它的 *Infrared\_task* 得以执行一个循环并进入延时阻塞状态。

延时时间到，*Car\_task* 解除阻塞，获得 CPU 使用权。接下来小车控制和红外检测任务轮流使用 CPU，完成巡线目标。

### 3.4.2 队列与通信的原理说明

在 FreeRTOS 中，队列被用于任务与任务之间或是中断与任务之间的信息传递。队列实质上是链表，每一个链表项大小相同、至多记录一条信息。消息一般遵循先进先出的规则进出列表。对于一个队列来说，任何一个任务都可以向其传递信息（其实是记录信息的地址），实质是将传递内容保存在链表项中。同样，任何一个任务都可以从该队列中读取信息。被读取过的信息根据程序设置可以被删除也可以被保留以供其他任务读取。

在 *Start\_task* 中，消息队列 *Message\_task* 被创建。只要队列未满，CPU 每执行一次 *Infrared\_task*，就会有一则红外探测消息被发送至队列中。若队列已满，*Infrared\_task* 就放弃发送这则消息，将 CPU 使用权交给 *Car\_task*。只要队列未空，CPU 每执行一次 *Car\_task*，队列开头的消息就会被该任务取出（读取后不再保留在队列中）。若是队列为空，则 *Car\_task* 进入阻塞态，直至有消息进入队列中。

## 4. 总结与展望

### 4.1 总结

本项目使用 Microbit 小车搭载 RISC-V SiFive Learn Inventor 芯片，实现了小车的循迹功能，通过调用 PWM 函数实现对小车速度的控制。为了实现多线程控制，我们引入了 FreeRTOS，最终达到了多程序并行执行的效果。在本项目报告中，我们详细介绍了项目实施的过程并展示了实验结果。

### 4.2 展望

通过本项目，我们对 RISC-V 芯片架构有了充分的了解，我们将在后续的工作中进一步完成智能小车的循迹避障功能，并尝试使用其他传感器，如激光雷达，深度摄像头等，实现对小车的控制