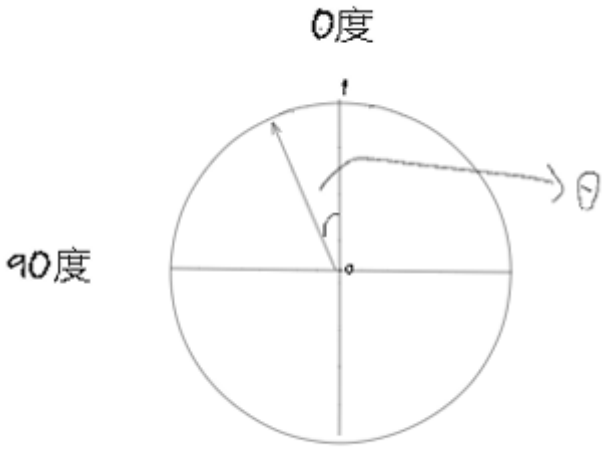
rk3568Android11小车笔记

1.设计思想

以方向轮盘正前方为起点，往**逆时针方向角度angle递增**，**作为方向direction**

以方向盘圆心为起点，**远离圆心speed递增**，**作为速度speed**



2.设备树

使用四个GPIO引脚,两个PWM引脚

```
/ {
car: car {
        compatible = "mycar";

        //I2C3_SDA_M0
        my_gpio1: gpio1_a0 {
            compatible = "mygpio";
            my-gpios = <&gpio1 RK_PA0 GPIO_ACTIVE_HIGH>;
            pinctrl-names = "default";
            pinctrl-0 = <&my_gpio1_a0_ctrl>;
        };

        //I2C3_SCL_M0
        my_gpio2: gpio1_a1 {
            compatible = "mygpio";
            my-gpios = <&gpio1 RK_PA1 GPIO_ACTIVE_HIGH>;
            pinctrl-names = "default";
            pinctrl-0 = <&my_gpio1_a1_ctrl>;
        };

        //PWM1_M0
        my_gpio3: gpio0_c0 {
```

```
            compatible = "mygpio";
            my-gpios = <&gpio0 RK_PC0 GPIO_ACTIVE_HIGH>;
            pinctrl-names = "default";
            pinctrl-0 = <&my_gpio0_c0_ctrl>;
        };

        //PWM2_M0
        my_gpio4: gpio0_c1 {
            compatible = "mygpio";
            my-gpios = <&gpio0 RK_PC1 GPIO_ACTIVE_HIGH>;
            pinctrl-names = "default";
            pinctrl-0 = <&my_gpio0_c1_ctrl>;
        };

        //PWM12_M1
        my_pwm1{
            compatible = "mypwm";
            pwms = <&pwm12 0 20000000 1>;
        };


        //PWM13_M1
        my_pwm2{
            compatible = "mypwm";
            pwms = <&pwm13 0 20000000 1>;
        };
    };
};
```

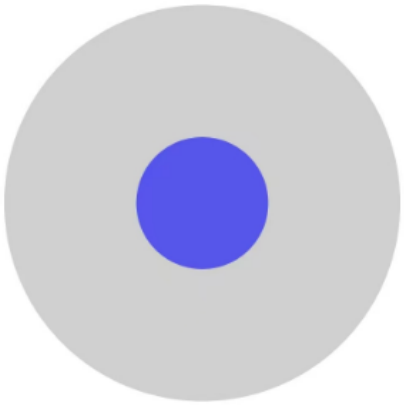
引脚复用设置

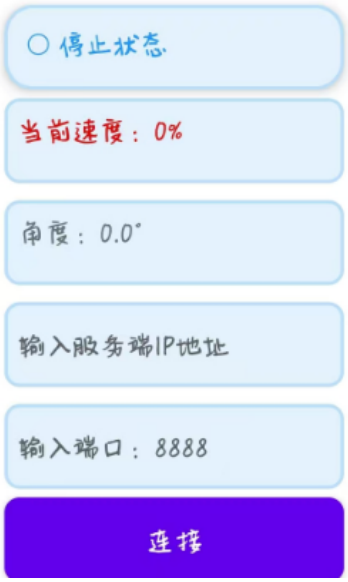
```
&pinctrl {

mygpio{
        my_gpio1_a0_ctrl:my-gpio-a0-ctrl{
            rockchip,pins = <1 RK_PA0 RK_FUNC_GPIO &pcfg_pull_none>;
        };
        my_gpio1_a1_ctrl:my-gpio-a1-ctrl{
            rockchip,pins = <1 RK_PA1 RK_FUNC_GPIO &pcfg_pull_none>;
        };
        my_gpio0_c0_ctrl:my-gpio-c0-ctrl{
            rockchip,pins = <0 RK_PC0 RK_FUNC_GPIO &pcfg_pull_none>;
        };
        my_gpio0_c1_ctrl:my-gpio-c1-ctrl{
            rockchip,pins = <0 RK_PC1 RK_FUNC_GPIO &pcfg_pull_none>;
        };
    };

&pwm12{
    status = "okay";
    pinctrl-names = "active";
    pinctrl-0 = <&pwm12m1_pins>;
};
```

```
&pwm13{
    status = "okay";
    pinctrl-names = "active";
    pinctrl-0 = <&pwm13m1_pins>;
};
};
```

3.Android客户端设计

Android客户端传递两个参数，使用socket通信发送angle和speed到rk3568服务端AndroidJNIapp





4.rk3568服务端设计

rk3568服务端接收到数据后，使用controlCar方法 调用ioctl传递angle和speed到驱动层

```
public native void controlCar(int angle, int speed);
```

rk358AndroidJNI的cpp代码

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <asm-generic/fcntl.h>
#include <fcntl.h>
#include <unistd.h>

int fd = 0;

extern "C" JNIEXPORT jint JNICALL
Java_com_example_carjni_MainActivity_MyDeviceOpen(
        JNIEnv* env,
        jobject /* this */) {
    fd = open("/dev/mydevice", O_RDWR | O_NDELAY | O_NOCTTY);
```

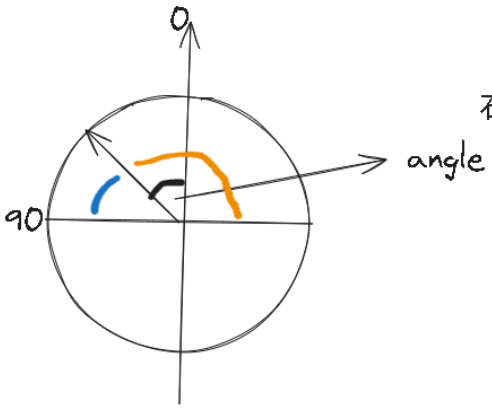
```
    if (fd < 0) {
        __android_log_print(ANDROID_LOG_INFO, "serial", "open error");
    }else {
        __android_log_print(ANDROID_LOG_INFO, "serial", "open success fd=%d",fd);
    }
    return 0;
}

extern "C" JNIEXPORT jint JNICALL
Java_com_example_carjni_MainActivity_MyDeviceClose(
        JNIEnv* env,
        jobject /* this */) {
    if (fd > 0) {
        close(fd);
    }
    return 0;
}

extern "C" JNIEXPORT void JNICALL
Java_com_example_carjni_MainActivity_controlCar(JNIEnv *env, jobject, jint angle,
jint speed) {
    ioctl(fd, speed, angle);
}
```

5.驱动层设计

方向控制思想

非线性变化，转弯灵敏

例如左转：



若左转，求左方向所占当前方向份额与
右方向所占当前方向份额的角度比值，也就是

左轮速度：

speed * (90-angle)/(90+angle)

右轮速度：

speed

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/platform_device.h>
#include <linux/mod_devicetable.h>
#include <linux/of.h>
```

```c
#include <linux/gpio/consumer.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/pwm.h>

#define MYDEVICE_NAME "mydevice"
#define MAX_GPIO_NODES 4 // 明确最大GPIO数量
#define MAX_PWM_NUM 2    // 最大PWM数量

static struct gpio_desc *mygpio[MAX_GPIO_NODES]; // GPIO描述符数组
static struct pwm_device *mypwm[MAX_PWM_NUM];// PWM描述符数组


// 字符设备相关
static dev_t mydevice_dev;
static struct cdev mydevice_cdev;
static struct class *mydevice_class;

//speed:0-100
/**
 * 控制速度的函数
 *
 * 该函数通过调整PWM（脉冲宽度调制）信号的占空比来控制速度
 * 它根据输入的速度参数计算出对应的高电平时间，并设置PWM设备
 *
 * @param speed 速度值，表示所需的速动大小
 * @param mypwm 指向PWM设备的结构体指针，用于配置PWM信号
 *
 * @return 返回0表示成功，非零表示失败
 */
void contrl_speed(int speed,struct pwm_device *mypwm){

    // 定义高电平时间变量
    int highTime;

    //该代码通过四舍五入计算speed乘以20000000后除以100的商，结果赋值给highTime。
    //DIV_ROUND_CLOSEST是实现了"最接近整数"除法算法的宏，用于将除法结果精确到整数而非取整。
    // 通过将速度转换为高电平时间，实现对速度的控制
    highTime = DIV_ROUND_CLOSEST(speed * 20000000, 100);

    // 设置PWM调速,20ms内高电平所占的时间
    // 设置PWM调速
    // 根据计算出的高电平时间和固定的周期时间配置PWM设备
    pwm_config(mypwm, highTime, 20000000);
}


//控制前进和后退方向
void contrl_direction(int angle){

    if((angle >=0 && angle <= 90) || (angle >= 270 && angle <= 360)){//前进方向
        printk("forward is up\n");
        gpiod_set_value(mygpio[0], 1);
        gpiod_set_value(mygpio[1], 0);
        gpiod_set_value(mygpio[2], 1);
```

```
        gpiod_set_value(mygpio[3], 0);
    }else if(angle > 90 && angle < 270){//后退方向
        printk("forward is down\n");
        gpiod_set_value(mygpio[0], 0);
        gpiod_set_value(mygpio[1], 1);
        gpiod_set_value(mygpio[2], 0);
        gpiod_set_value(mygpio[3], 1);
    }
}

void EAstop(void){
    int i;
    for(i=0;i<MAX_PWM_NUM;i++){
        pwm_disable(mypwm[i]);//关闭PWM
    }
}

void EAstart(void){
    int  i;
    for(i=0;i<MAX_PWM_NUM;i++){
        pwm_enable(mypwm[i]);//使能PWM
    }
}s

static int mydriver_open(struct inode *inode, struct file *file)
{
    printk("mydevice open\n");
    return 0;
}

static int mydriver_release(struct inode *inode, struct file *file)
{
    // 关闭PWM
    pwm_disable(mypwm[0]);
    pwm_disable(mypwm[1]);
    printk("mydevice release\n");
    return 0;
}

static long mydriver_ioctl(struct file *file, unsigned int cmd, unsigned long
arg)
{
    int speed = cmd;
    int angle = arg;
    int rotation;

    //speed为速度，0-100，angle为角度，0-360度
    printk("speed is %d\n",speed);
    printk("angle is %d\n",angle);

    //速度=0
    if(speed == 0){
        //停止
        EAstop();
    }
    else if(speed > 0){
```

```
        //启用PWM
        EAstart();

        //左转
        if(angle >= 0 && angle <= 90){
            printk("left\n");
            //控制方向
            contrl_direction(angle);

            rotation = (90 - angle) * 1000 / (90 + angle);

            //调整左轮速度
            contrl_speed(speed * rotation / 1000, mypwm[0]);
            contrl_speed(speed,mypwm[1]);
        }

        //右转
        if(angle >= 270 && angle <= 360){
            printk("right\n");
            //控制方向
            contrl_direction(angle);

            rotation = (angle - 270) * 1000/ (450 - angle);

            //调整右轮速度
            contrl_speed(speed,mypwm[0]);
            contrl_speed(speed * rotation / 1000,mypwm[1]);

        }

        //左后转
        if(angle > 90 && angle <= 180){
            printk("left|down\n");
            //控制方向
            contrl_direction(angle);

            rotation = (angle - 90) * 1000 / (270 - angle);

            //调整左轮速度
            contrl_speed(speed * rotation / 1000,mypwm[0]);
            contrl_speed(speed,mypwm[1]);

        }

        //右后转
        if(angle > 180 && angle < 270){
            printk("right|down\n");
            //控制方向
            contrl_direction(angle);

            rotation = (270 -angle) * 1000 / (angle - 90);

            //调整右轮速度
            contrl_speed(speed,mypwm[0]);
            contrl_speed(speed * rotation / 1000,mypwm[1]);
        }
```

```
    }
    return 0;
}

static struct file_operations mydevice_fops = {
    .owner = THIS_MODULE,
    .open = mydriver_open,
    .release = mydriver_release,
    .unlocked_ioctl = mydriver_ioctl,
};

int mydriver_probe(struct platform_device *pdev)
{
    struct device_node *car_node = pdev->dev.of_node;
    struct device_node *child;
    int i = 0;
    int j = 0;
    int ret;
    int num;

    printk("mydriver_probe!\n");

    if (!car_node) {
        printk("Failed to find car node\n");
        return -ENODEV;
    }

    // 遍历子节点，打印GPIO和PWM节点信息，初始化,GPIO,PWM
    for_each_child_of_node(car_node, child) {
        printk("Child node: %s\n", child->name);

        // 检查节点是否属于 GPIO
        if (of_device_is_compatible(child, "mygpio")) {
            if (i >= MAX_GPIO_NODES) {
                printk("Too many GPIO nodes, skip %s\n", child->name);
                continue;
            }
            // 获取 GPIO
            mygpio[i] = devm_gpiod_get_from_of_node(&pdev->dev, child, "my-gpios",
0, GPIOD_OUT_LOW, NULL);
            if (IS_ERR(mygpio[i])) {
                printk("Get GPIO failed for %s\n", child->name);
                mygpio[i] = NULL;
            } else {
                // 设置 GPIO 方向为输出
                gpiod_direction_output(mygpio[i], 0);
                // 获取 GPIO 编号
                num = desc_to_gpio(mygpio[i]);
                printk("GPIO %s num: %d\n", child->name, num);
            }
            i++;
        }
        // 检查节点是否属于 PWM
        else if (of_device_is_compatible(child, "mypwm")) {
            if (j >= MAX_PWM_NUM) {
                printk("Too many PWM nodes, skip %s\n", child->name);
```

```
                continue;
            }
            // 获取 PWM（使用索引或唯一标识）
            mypwm[j] = devm_of_pwm_get(&pdev->dev, child, NULL);
            if (IS_ERR(mypwm[j])) {
                printk("Get PWM failed for %s\n", child->name);
                mypwm[j] = NULL;
            } else {
                // 配置 PWM 参数（周期 20ms）
                pwm_config(mypwm[j], 10000000, 20000000); // 周期20ms，占空比50%
                // 设置 PWM 极性（正极）
                pwm_set_polarity(mypwm[j], PWM_POLARITY_NORMAL);
                ret = pwm_enable(mypwm[j]); // 直接在此处启用
                if (ret < 0) {
                    printk("Failed to enable PWM %d\n",j);
                }
            }
            j++;
        }
    }

    // 注册字符设备（带错误处理）
    ret = alloc_chrdev_region(&mydevice_dev, 0, 1, MYDEVICE_NAME);
    if (ret < 0) {
        printk("Failed to allocate chrdev\n");
        return ret;
    }

    cdev_init(&mydevice_cdev, &mydevice_fops);
    ret = cdev_add(&mydevice_cdev, mydevice_dev, 1);
    if (ret < 0) {
        printk("Failed to add cdev\n");
        goto chrdev_err;
    }

    mydevice_class = class_create(THIS_MODULE, MYDEVICE_NAME);
    if (IS_ERR(mydevice_class)) {
        ret = PTR_ERR(mydevice_class);
        printk("Failed to create class\n");
        goto class_err;
    }

    device_create(mydevice_class, NULL, mydevice_dev, NULL, MYDEVICE_NAME);
    return 0;

class_err:
    cdev_del(&mydevice_cdev);
chrdev_err:
    unregister_chrdev_region(mydevice_dev, 1);
    return ret;
}

int mydriver_remove(struct platform_device *dev)
{
    printk("mydriver_remove!\n");
```

```
    // 卸载字符设备
    device_destroy(mydevice_class, mydevice_dev);
    class_destroy(mydevice_class);
    cdev_del(&mydevice_cdev);
    unregister_chrdev_region(mydevice_dev, 1);

    return 0;
}

// 定义一个常量结构体，用于平台设备的识别
const struct platform_device_id mydriver_id_table = {
    .name = "mydevice", // 设备名称，用于匹配特定的设备
};

const struct of_device_id of_match_table_id[] = {
    {
        .compatible = "mycar",
    },
    {}};

// 定义一个平台驱动结构体，用于注册平台驱动程序
struct platform_driver platform_driver_test = {
    // 指向探针函数的指针，当检测到设备时自动调用此函数进行初始化
    .probe = mydriver_probe,
    // 指向移除函数的指针，当设备被移除时自动调用此函数进行清理
    .remove = mydriver_remove,
    // 驱动结构体，包含驱动的元数据信息
    .driver = {
        // 驱动的名称，用于在系统中标识此驱动
        .name = "mydevice",
        // 指明驱动的拥有者，通常为THIS_MODULE，用于模块管理
        .owner = THIS_MODULE,
        // 设备树匹配表，用于匹配设备树中的设备
        .of_match_table = of_match_table_id,
    },
    // 设备ID表，定义了此驱动支持的设备ID
    .id_table = &mydriver_id_table,
};

module_platform_driver(platform_driver_test);

// 指定模块的许可证为GPL
MODULE_LICENSE("GPL");
// 指定模块的作者信息
MODULE_AUTHOR("XYY");
// 指定模块的版本号
MODULE_VERSION("V0.1");
```

6.终端启动/停止app

```
am start -n com.example.carjni/.MainActivity

am force-stop com.example.carjni
```