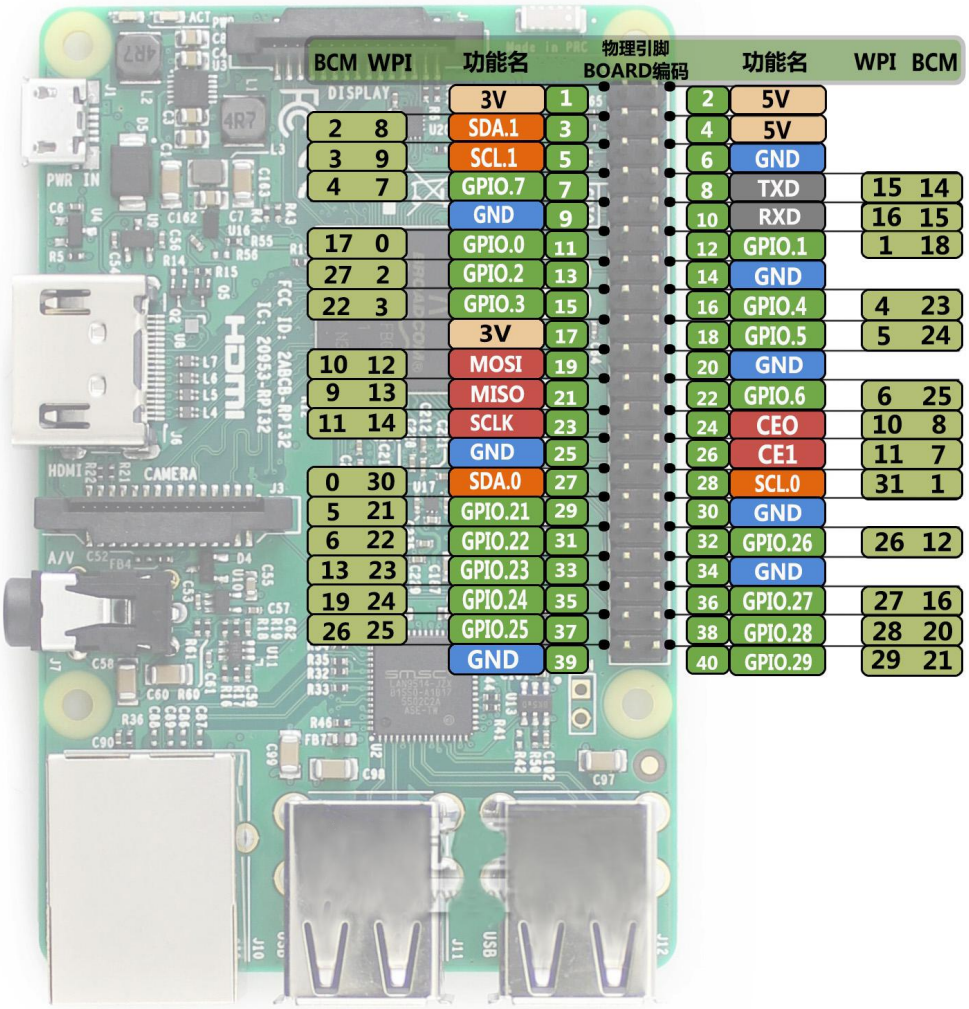
树莓派 wiringPi 引脚



树莓派 wiringPi 库详解

wiringPi 是一个很棒的树莓派 IO 控制库，使用 C 语言开发，提供了丰富的接口：GPIO 控制，中断，多线程，等等。java 的 pi4j 项目也是基于 wiringPi 的，我最近也在看源代码，到时候整理好了会放出来的。

下面开始 wiringPi 之旅吧！

安装

进入 wiringPi 的 github (https://git.drogon.net/?p=wiringPi;a=summary) 下载安装包。点击页面的第一个链接的右边的 snapshot,下载安装压缩包。

然后进入安装包所在的目录执行以下命令：

```
>tar xfz wiringPi-98bcb20.tar.gz   //98bcb20 为版本标号，可能不同>cd
wiringPi-98bcb20

>./build
```

验证 wiringPi 的是否安装成功，输入 gpio -v，会在终端中输出相关 wiringPi 的信息。否则安装失败。

编译 和运行

假如你写了一个 LEDtest.c 的项目，则如下。

编译：

```
g++ -Wall -o LEDtest LEDtest.cpp  -lwiringPi        //使用 C++编程 , -Wall 是为了使能所有警告，以便发现程序中的问题
```

```
gcc -Wall -o LEDtest LEDtest.c   -lwiringPi         //使用 C 语言编程
```

运行：

```
sudo ./LEDtest
```

查看引脚编号表格

使用如下控制台下命令

```
> gpio readall
```

也可以查看下面的图。

注意：查看时，将树莓派的 USB 接口面对自己，这样看才是正确的。

树莓派 40Pin 引脚对照表

wiringPi 编码	BCM 编码	功能名	物理引脚 BOARD编码		功能名	BCM 编码	wiringPi 编码
		3.3V	1	2	5V		
8	2	SDA.1	3	4	5V		
9	3	SCL.1	5	6	GND		
7	4	GPIO.7	7	8	TXD	14	15
		GND	9	10	RXD	15	16
0	17	GPIO.0	11	12	GPIO.1	18	1
2	27	GPIO.2	13	14	GND		
3	22	GPIO.3	15	16	GPIO.4	23	4
		3.3V	17	18	GPIO.5	24	5
12	10	MOSI	19	20	GND		
13	9	MISO	21	22	GPIO.6	25	6
14	11	SCLK	23	24	CE0	8	10
		GND	25	26	CE1	7	11
30	0	SDA.0	27	28	SCL.0	1	31
21	5	GPIO.21	29	30	GND		
22	6	GPIO.22	31	32	GPIO.26	12	26
23	13	GPIO.23	33	34	GND		
24	19	GPIO.24	35	36	GPIO.27	16	27
25	26	GPIO.25	37	38	GPIO.28	20	28
		GND	39	40	GPIO.29	21	29

表格由树莓派实验室绘制 http://shumeipai.nxez.com

wiringPi 库 API 大全

在使用 wiringPi 库时，你需要包含头文件 #include<wiringPi.h>。凡是写 wiringPi 的程序，都包含这个头文件。

硬件初始化函数

使用 wiringPi 时，你必须在执行任何操作前初始化树莓派，否则程序不能正常工作。

可以调用下表函数之一进行初始化，它们都会返回一个 int ， 返回 -1 表示初始化失败。

int wiringPiSetup (void)	返回:执行状态，-1 表示失败	当使用这个函数初始化树莓派引脚时，程序使用的是 wiringPi 引脚编号表。引脚的编号为 0~16 需要 root 权限
int wiringPiSetupGpio (void)	返回执行状态，-1 表示失败	当使用这个函数初始化树莓派引脚时，程序中使用的是 BCM GPIO 引脚编号表。 需要 root 权限
wiringPiSetupPhys(void)	不常用，不做介绍	/
wiringPiSetupSys (void) ;	不常用，不做介绍	/

通用 GPIO 控制函数

void pinMode (int pin, int mode)	pin：配置的引脚 mode:指定引脚的 IO 模式 可取的值：INPUT、OUTPUT、PWM_OUTPUT，GPIO_CLOCK	作用：配置引脚的 IO 模式 注意： 只有 wiringPi 引脚编号下的 1 脚（BCM 下的 18 脚） 支持 PWM 输出 只有 wiringPi 编号下的 7（BCM 下的 4 号）支持 GPIO_CLOCK 输出
void digitalWrite (int pin, int value)	pin：控制的引脚 value：引脚输出的电平值。 可取的值：HIGH，LOW 分别代表高低电平	让对一个已近配置为输出模式的 引脚 输出指定的电平信号
int digitalRead (int pin)	pin：读取的引脚 返回：引脚上的电平，可以是 LOW HIGH 之一	读取一个引脚的电平值 LOW HIGH ，返回
void analogWrite(int pin, int value)	pin:引脚 value：输出的模拟量	模拟量输出 树莓派的引脚本身是不支持 AD 转换的，也就是不能使用模拟量的 API， 需要增加另外的模块
int analogRead (int pin)	pin：引脚 返回：引脚上读取的模拟量	模拟量输入 树莓派的引脚本身是不支持 AD 转换的，也就是不能使用模拟量的 API， 需要增加另外的模块
void pwmWrite (int pin, int value)	pin：引脚 value：写入到 PWM 寄存器的值，范围在 0~1024 之间。	输出一个值到 PWM 寄存器，控制 PWM 输出。pin 只能是 wiringPi 引脚编号下的 1 脚（BCM 下的 18 脚）
void pullUpDnControl (int pin, int pud)	pin：引脚	对一个设置 IO 模式为 INPUT 的输入引脚设置拉电阻模式。

	pud：拉电阻模式 可取的值：PUD-OFF 关闭拉电阻 PUD_DOWN 引脚电平拉到3.3v PUD_UP 引脚电平拉到0v 接地	与 Arduino 不同的是，树莓派支持的拉电阻模式更丰富。 树莓派内部的拉电阻达 50K 欧姆

LED 闪烁程序

```
#include<iostream>
#include<cstdlib>
#include<wiringPi.h>

const int LEDpin = 1;

int main()
{
     if(-1==wiringPiSetup())
     {
          cerr<<"setup error\n";
          exit(-1);
     }
     pinMode(LEDpin,OUTPUT);


     for(size_t i=0;i<10;++i)
     {
              digitalWrite(LEDpin,HIGH);
              delay(600);
              digitalWrite(LEDpin,LOW);
              delay(600);
```

```
    }


    cout<<"------------bye-------------"<<endl;
    return 0;
}
```

PWM 输出控制 LED 呼吸灯的例子

```
#include<iostream>
#include<wiringPi.h>
#include<cstdlib>
using namespace std;



const int PWMpin = 1;   //只有 wiringPi 编号下的 1 脚（BCM 标号下的 18 脚）支持
void setup();

int main()
{

    setup();
    int val = 0;
    int step = 2;
    while(true)
    {
        if(val>1024)
        {
            step = -step;
            val = 1024;
        }
        else if(val<0)
        {
```

```
            step = -step;
            val = 0;
        }

        pwmWrite(PWMpin,val);
        val+=step;
        delay(10);
    }

    return 0;
}

void setup()
{
    if(-1==wiringPiSetup())
    {
        cerr<<"setup error\n";
        exit(-1);
    }
    pinMode(PWMpin,PWM_OUTPUT);
}
```

时间控制函数

unsigned int millis (void)	这个函数返回 一个 从你的程序执行 wiringPiSetup 初始化函数（或者 wiringPiSetupGpio ） 到 当前时间 经过的 毫秒数。 返回类型是 unsigned int，最大可记录 大约 49 天的毫秒时长。
unsigned int micros (void)	这个函数返回 一个 从你的程序执行 wiringPiSetup 初始化函数（或者 wiringPiSetupGpio ） 到 当前时间 经过的 微秒数。 返回类型是 unsigned int，最大可记录 大约 71 分钟的时长。
void delay (unsigned int	将当前执行流暂停 指定的毫秒数。因为 Linux 本身是多线程的，所以实

howLong)	际暂停时间可能会长一些。参数是 unsigned int 类型，最大延时时间可达 49 天
void delayMicroseconds (unsigned int howLong)	将执行流暂停 指定的微秒数（1000 微秒 = 1 毫秒 = 0.001 秒）。 因为 Linux 本身是多线程的，所以实际暂停时间可能会长一些。参数是 unsigned int 类型，最大延时时间可达 71 分钟

中断

wiringPi 提供了一个中断处理注册函数，它只是一个注册函数，并不处理中断。他无需 root 权限。

int wiringPiISR (int pin, int edgeType, void (*function)(void))	返回值：返回负数则代表注册失败 pin：接受中断信号的引脚 edgeType：触发的方式。 INT_EDGE_FALLING：下降沿触发 INT_EDGE_RISING：上升沿触发 INT_EDGE_BOTH ：上下降都会触发 INT_EDGE_SETUP：编程时用不到。 function：中断处理函数的指针，它是一个无返回值，无参数的函数。	注册的函数会在中断发生时执行 和 51 单片机不同的是：这个注册的中断处理函数会和 main 函数并发执行（同时执行，谁也不耽误谁） 当本次中断函数还未执行完毕，这个时候树莓派又触发了一个中断，那么这个后来的中断不会被丢弃，它仍然可以被执行。但是 wiringPi 最多可以跟踪并记录后来的仅仅 1 个中断，如果不止 1 个，则他们会被忽略，得不到执行。

通过 1 脚检测 因为按键按下引发的 下降沿，触发中断，反转 11 控制的 LED

```
#include<iostream>
#include<wiringPi.h>
#include<cstdlib>
using namespace std;

void ButtonPressed(void);
void setup();

/*******************************/
const int LEDPin = 11;
const int ButtonPin = 1;
/*******************************/

int main()
{

    setup();

    //注册中断处理函数
    if(0>wiringPiISR(ButtonPin,INT_EDGE_FALLING,ButtonPressed))
    {
        cerr<<"interrupt function register failure"<<endl;
        exit(-1);
    }



    while(1)
    ;
```

```
    return 0;
}

void setup()
{
    if(-1==wiringPiSetup())
    {
        cerr<<"wiringPi setup error"<<endl;
        exit(-1);
    }

    pinMode(LEDPin,OUTPUT);    //配置 11 脚为控制 LED 的输出模式
    digitalWrite(LEDPin,LOW);  //初始化为低电平

    pinMode(ButtonPin,INPUT);           //配置 1 脚为输入
    pullUpDnControl(ButtonPin,PUD_UP);  //将 1 脚上拉到 3.3v

}


//中断处理函数：反转 LED 的电平
void ButtonPressed(void)
{

    digitalWrite(LEDPin,  (HIGH==digitalRead(LEDPin))?LOW:HIGH );

}
```

多线程

wiringPi 提供了简单的 Linux 系统下的通用的 Posix threads 线程库接口来支持并发。

int piThreadCreate(name)	name:被包装的线程执行函数
	返回：状态码。返回 0 表示成功启动，反之失败。
	源代码：
	int piThreadCreate (void *(*fn)(void *)) { pthread_t myThread ; return pthread_create (&myThread, NULL, fn, NULL) ; }
		包装一个用 PI_THEEAD 定义的函数为一个线程，并启动这个线程。
		首先你需要通过以下方式创建一个特特殊的函数，这个函数中的代码就是在新的线程中将执行的代码。，myTread 是你自己线程的名字，可自定义。
		PI_THREAD (myThread) { //在这里面写上的代码会和主线程并发执行。 }
		在 wiringPi.h 中，我发现这样一个宏定义：#define PI_THREAD(X) void *X (void *dummy)
		那么，被预处理后我们写的线程函数会变成下面这个样子，请注意返回值，难怪我每次写都会警告，因为没有返回一个指针，那么，以后注意返回 NULL，或者 (void*)0
		void *myThread (void *dummy) { //在这里面写上的代码会和主线程并发执行。 }

piLock(int keyNum)	keyNum:0-3 的值，每一个值代表一把锁	使能同步锁。wiringPi 只提供了 4 把锁，也就是 keyNum 只能取 0~3 的值，官方认为有这 4 把锁就够了。 keyNum：0,1,2,3 每一个数字就代表一把锁。

		源代码：
		void piLock (int keyNum)
		{
		pthread_mutex_lock (&piMutexes [keyNum]) ;
		}

piUnlock(int keyNum)	keyNum:0-3 的值，每一个值代表一把锁	解锁，或者说让出锁。
		源代码：
		void piUnlock (int key)
		{
		pthread_mutex_unlock (&piMutexes [key]) ;
		}

int piHiPri (int priority)	priority：优先级指数，0~99 返回值：0，成功 -1:，失败	设定线程的优先级，设定线程的优先级变高，不会使程序运行加快，但会使这个线程获得相当更多的时间片。priority 是相对的。比如你的程序只用到了主线程， 和另一个线程 A，主线程设定优先级为 1，A 线程设定为 2，那也代表 A 比 main 线程优先级高。

凡是涉及到多线程编程，就会涉及到线程安全的问题，多线程访问同一个数据，需要使用同步锁来保障数据操作正确性和符合预期。

当 A 线程锁上 锁 S 后，其他共用这个锁的竞争线程，只能等到锁被释放，才能继续执行。

成功执行了 piLock 函数的线程将拥有这把锁。其他线程想要拥有这把锁必须等到这个线程释放锁，也就是这个线程执行 piUnlock 后。

同时要扩展的知识是：volatile 这个 C/C++中的关键字，它请求编译器不缓存这个变量的数据，而是每次都从内存中读取。特别是在多线程下共享放变量，必须使用 volatile 关键字声明才是保险的。

softPwm,软件实现 PWM

树莓派硬件上支持的 PWM 输出的引脚有限，为了突破这个限制，wiringPi 提供了软件实现的 PWM 输出 API。

需要包含头文件：#include <softPwm.h>

编译时需要添 pthread 库链接 -lpthread

int softPwmCreate (int pin, int initialValue, int pwmRange)	pin：用来作为软件 PWM 输出的引脚 initalValue：引脚输出的初始值 pwmRange：PWM 值的范围上限 建议使用 100. 返回：0 表示成功。	使用一个指定的 pin 引脚创建一个模拟的 PWM 输出引脚
void softPwmWrite (int pin, int value)	pin：通过 softPwmCreate 创建的引脚 value：PWM 引脚输出的值	更新引脚输出的 PWM 值

串口通信

使用时需要包含头文件：#include <wiringSerial.h>

int serialOpen (char *device, int baud)	device:串口的地址，在 Linux 中就是设备所在的目录。 默认一般是"/dev/ttyAMA0",我的是这样的。 baud：波特率 返回：正常返回文件描述符，否则返回-1 失败。	打开并初始串口
void serialClose (int fd)	fd：文件描述符	关闭 fd 关联的串口
void serialPutchar (int fd, unsigned char c)	fd:文件描述符 c:要发送的数据	发送一个字节的数据到串口
void serialPuts (int fd, char *s)	fd：文件描述符 s：发送的字符串，字符串要以'\0'结尾	发送一个字符串到串口
void serialPrintf (int fd, char *message, …)	fd：文件描述符 message：格式化的字符串	像使用 C 语言中的 printf 一样发送数据到串口
int serialDataAvail (int fd)	fd：文件描述符 返回：串口缓存中已经接收的，可读取的字节数，-1 代表错误	获取串口缓存中可用的字节数。
int serialGetchar (int fd)	fd：文件描述符 返回：读取到的字符	从串口读取一个字节数据返回。 如果串口缓存中没有可用的数据，则会等待 10 秒，如果 10 后还有没，返回-1 所以，在读取前，做好通过

		serialDataAvail 判断下。
void serialFlush (int fd)	fd：文件描述符	刷新，清空串口缓冲中的所有可用的数据。
*size_t write (int fd, const void *buf, size_t count)	fd：文件描述符 buf：需要发送的数据缓存数组 count:发送 buf 中的前 count 个字节数据 返回：实际写入的字符数，错误返回-1	这个是 Linux 下的标准 IO 库函数，需要包含头文件#include <unistd.h> 当要发送到的数据量过大时，wiringPi 建议使用这个函数。
*size_t read(int fd, void *buf , size_t count);	fd：文件描述符 buf：接受的数据缓存的数组 count:接收的字节数. 返回：实际读取的字符数。	这个是 Linux 下的标准 IO 库函数，需要包含头文件#include <unistd.h> 当要接收的数据量过大时，wiringPi 建议使用这个函数。

初次使用树莓派串口编程，需要配置。

```
/* 修改 cmdline.txt 文件 */
>cd /boot/
>sudo vim cmdline.txt

删除【】之间的部分
dwc_otg.lpm_enable=0 【console=ttyAMA0,115200】 kgdboc=ttyAMA0,115200
console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait

/*修改 inittab 文件 */
>cd /etc/
>sudo vim inittab

注释掉最后一行内容:，在前面加上 # 号
#T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

```
sudo reboot 重启
```

下面是双机通信的一个例子

C51 代码，作为串口通信的接发送。

```
#include<reg52.h>
#include"serial.h"

/**********function***************/
bit isOpenPressed(void);
bit isClosePressed(void);
void delay(unsigned int t);
/*********************************/

sbit closeButton = P2^0;   //与关闭按键相连的引脚
sbit openButton  = P2^1;   //与打开按键相连的引脚

void main(void)
{

     closeButton = 1;      //拉高
     openButton  = 1;      //拉高

     EA =1;             //打开总中断
     serial_init(9600);  //初始化 51 串口

     while(1)
     {
        if(isClosePressed())     //如果关闭按钮按下
        {
```

```
                serial_write(0);  //发送数据 0 给树莓派
              delay(10);
         }

         else if(isOpenPressed()) //如果打开按钮按下
         {
                serial_write(1);     //发送数据 1 给树莓派
              delay(10);
         }
      }
}



bit isOpenPressed(void)
{
       bit press =0;

       if(0==openButton)
       {
              delay(5);
            if(0==openButton)
            {
                while(!openButton)
                ;
                press = 1;
            }
       }

       return press;
}

bit isClosePressed(void)
{
       bit press =0;

       if(0==closeButton)
```

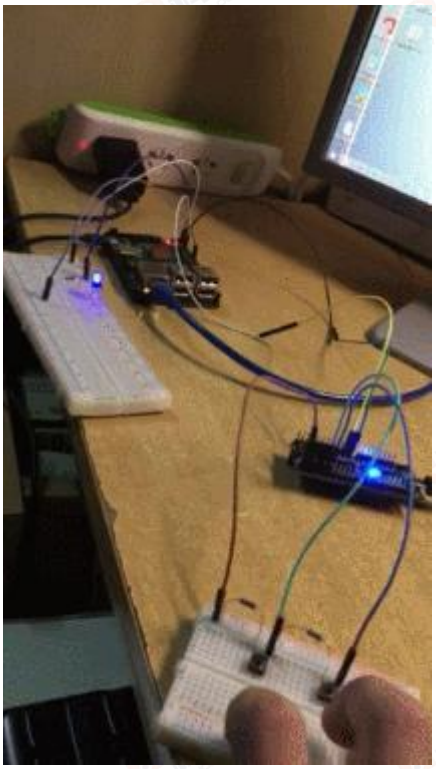
```
      {
             delay(5);
          if(0==closeButton)
          {
              while(!closeButton)
              ;
               press = 1;
          }
      }

      return press;
}



void delay(unsigned int t)
{
    unsigned int  i    ;
    unsigned char j;
    for(i = t;i>0;i--)
        for(j=120;j>0;j--)
            ;
}
```

树莓派代码，作为串口通信的接收方

```
#include<iostream>
#include<cstdlib>
#include<wiringPi.h>
#include<wiringSerial.h>
using namespace std;



void setup();
const int LEDPin = 11;
```

```
int main()
{
    setup();

    int fd; //Linux 的思想是：将一切 IO 设备，都看做 文件，fd 就是代表串口抽象出来的文件

    if((fd = serialOpen("/dev/ttyAMA0",9600))==-1)    //初始化串口，波特率 9600
    {

        cerr<<"serial open error"<<endl;
        exit(-1);

    }

    while(true)
    {

        if(serialDataAvail(fd) >= 1)    //如果串口缓存中有数据
        {
            int data = serialGetchar(fd);

            if(data==0)   //接受到 51 发送的 数据 0
            {
                // close led
                digitalWrite(LEDPin,LOW);

            }
            else if(data==1)  //接受到 51 发送的 数据 1
            {
                //open led
                digitalWrite(LEDPin,HIGH);
            }
        }
```

```
    }


    return 0;
}



void setup()
{
    if(-1==wiringPiSetup())
    {
        cerr<<"set up error"<<endl;
        exit(-1);
    }

    pinMode(LEDPin,OUTPUT);
    digitalWrite(LEDPin,HIGH);

}
```



Shift 移位寄存器 API

需要包含头文件 #include <wiringShift.h>

void shiftOut (uint8_t dPin, uint8_t cPin, uint8_t order, uint8_t val)	dPin：移位芯片的串行数据入口引脚，比如 74HC595 的 SER 脚 cPin：移位芯片的时钟引脚。如 74HC595 的 11 脚 order： LSBFIRST 先发送数据的低位 MSBFIRST 先发送数据的高位 val：要发送的 8 位数据	将 val 串化，通过芯片转化为并行输出 如常见的 74HC595
uint8_t shiftIn (uint8_t dPin, uint8_t cPin, uint8_t order)	同上。	将并行数据，通过芯片转化为串行输出。

用过 595 的都知道还有一个引脚：12 脚，Rpin，用于把移位寄存器中的数据更新到存储寄存器中，然后 wiringPi 的 API 中没有使用这个引脚。我建议使用的时候自己加上。

```
#include<iostream>
#include<wiringPi.h>
#include <wiringShift.h>
#include<cstdlib>
using namespace std;

const int SERpin = 1;   //serial data input
const int SCKpin = 2;   //shift register clock
const int RCKpin = 3;   // storage register clock
/************************/
```

```
void setup();

/************************/

int main()
{
    setup();

    for(int i=0;i<8;++i)
    {
        digitalWrite(RCKpin,LOW);

              shiftOut(SERpin,SCKpin,LSBFIRST,1<<i);
        digitalWrite(RCKpin,HIGH);

              delay(800);
    }
    return 0;

}



void setup()
{
    if(-1==wiringPiSetup())
    {
        cerr<<"setup error\n";
        exit(-1);
    }

    pinMode(SERpin,OUTPUT);
    pinMode(RCKpin,OUTPUT);
    pinMode(SCKpin,OUTPUT);
```

```
}
```

树莓派硬件平台特有的 API

并没有列全，我只是列出了相对来说有用的，其他的，都基本不会用到。

pwmSetMode (int mode)	mode：PWM 运行模式	设置 PWM 的运行模式。 pwm 发生器可以运行在 2 种模式下，通过参数指定： PWM_MODE_BAL ：树莓派默认的 PWM 模式 PWM_MODE_MS ：传统的 pwm 模式，
pwmSetRange (unsigned int range)	range，范围的最大值 0~range	设置 pwm 发生器的数值范围，默认是 1024
pwmSetClock (int divisor)		This sets the divisor for the PWM clock. To understand more about the PWM system, you'll need to read the Broadcom ARM peripherals manual.
piBoardRev (void)	返回：树莓派板子的版本编号 1 或者 2	/