第 5 章

单片机入门——Arduino

前面介绍章节的内容是以模拟电子技术为主，即处理的电压或电流等信号是随时间连续变化的。通常来讲，使用模拟电子技术实现的设备电路较为复杂。相比于模拟技术，应用数字电子技术则能使设备的体积缩小、功耗降低以及可靠性大幅提高。特别是集成电路的价格随着生产工艺的提高不断进步而越发低廉，数字电子技术的应用也越来越广泛。因此，本章及以后章节均对数字电子相关技术进行介绍。数字技术涉及的内容非常多，包括基础数字电子电路、单片机、PLD、FPGA 等。由于篇幅原因，本书不能对所有内容都进行详细介绍。又由于单片机相关技术应用的特别广泛，包括身边随处可见的家电、工业设备用的各种控制器、汽车电子相关产品以及航空航天系统、尖端武器等军工领域等，因此本书将重点介绍单片机相关技术，其他内容读者需自行查阅相关资料。

事实上，设计基于单片机的电子产品可以认为是设计一台微型、具有特殊功能的计算机，涉及软、硬件的各种知识，这也是初学者较难入门单片机的原因。通常来说，学习过程可以认为是一个从感性思维到理性思维的过程。利用 Arduino 开发单片机系统可以让使用者不用具备太多底层软硬件知识就能轻易上手，有利于让读者对单片机有一个感性的认识，非常适合初学者。因此本章不会介绍过多单片机底层知识，主要介绍如何使用 Arduino 进行一些验证性实验，带领读者入门单片机技术。

5.1 数字电子技术与模拟电子技术

本书第 1～4 章所介绍的为模拟电子技术，可以看出，模拟电子主要实现模拟信号的放大、滤波等功能，从而达到信号处理与能量转化的目的。在模拟电路中，晶体管一般工作在线性放大区域，当外界环境变化时，晶体管的放大特性会发生变化，从而影响信号传输的准确性，甚至是产生失真。因此，模拟电子技术的抗干扰能力与稳定性较差。

数字电子技术主要对离散信号进行处理，一般都采用二进制来表示数字信号。而二进制则可以利用元器件的两个稳定状态来表示，例如，在稳态下，可以让三极管处于饱和区和截止区，则在这两种状态下表现出的现象为电流的有、无或电压的高、低，这种有与无、高与低的状态分别可用二进制中的 1 与 0 进行表示。因此在数字电子电路中，其基本单元电路简单，对电路中各元件精度要求不很严格，允许元件参数有较大的分散性，只要能区分两种截然不同的状态即可。因此数字电子技术具有更好的稳定性、抗干扰性，更适合电路的集成化与小性化。

正是由于这些优点，数字电路得到了更广泛的应用。但是这并不代表数字电子技术可以完全取代模拟电子技术，例如，在大功率的功放电路、小信号放大电路等场景中，模拟电路更具优势。另外，由于数字电路中存在大量的跳变信号（例如，方波），相比与模拟电路，数字电路会产生大量的噪声。因此模拟电子技术与数字电子技术各有其优缺点，在实际应用中，往往需要模拟与数字相结合，充分发挥各自的特点。

5.2　初识单片机——Arduino

Arduino 是一类便捷灵活、方便上手的开源电子原型平台，可使用 C/C++语言进行开发。具有一套完整的开发生态链，包含一系列适用不同应用的开源硬件以及丰富的库函数，基于 Arduino 的电子系统开发更多是对库函数的使用，只需要注重应用层的逻辑，不需要了解底层寄存器就可完成复杂的应用，非常适合初学者学习单片机技术。不同 Arduino 硬件之间共用一套成熟的库函数以及对应的 API 接口，因此只需要精通一种 Arduino 硬件即可，本书是以 Arduino UNO 板（以下简称 Arduino 板）介绍 Arduino 的相关知识。

5.2.1　硬件基础

如图 5-1 所示为 Arduino UNO 板的引脚分配图，包含 14 个数字引脚、6 个模拟输入、电源插孔、USB 连接和 ICSP 插头。

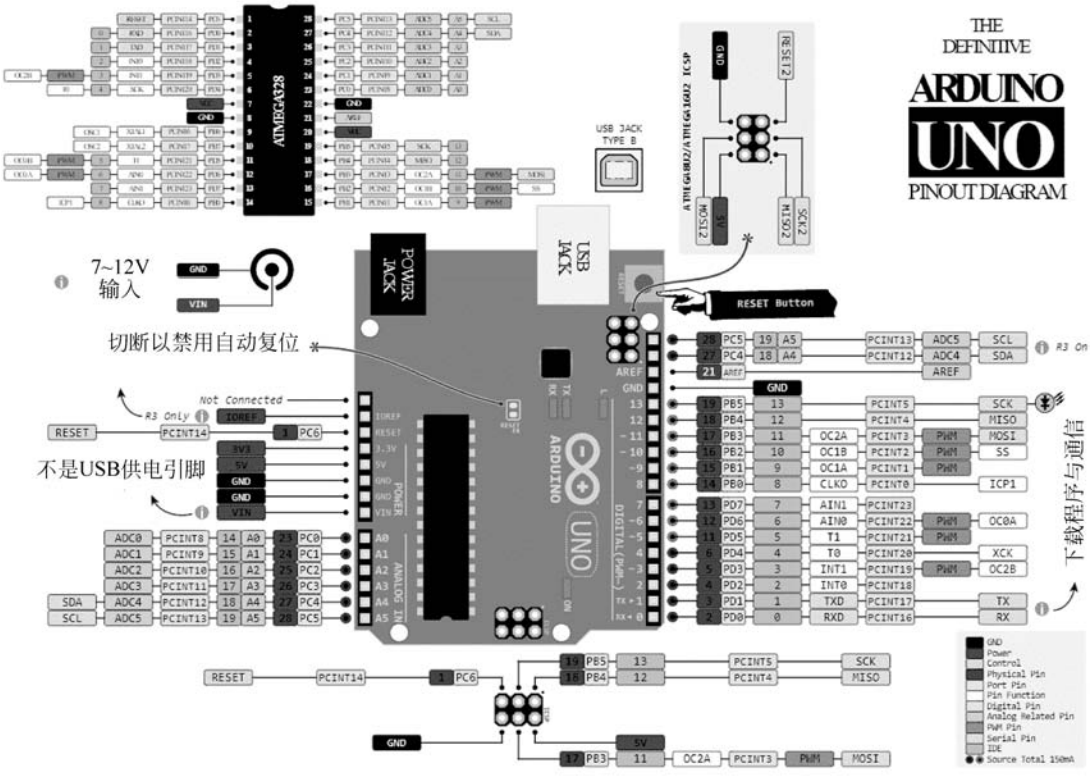


图 5-1　Arduino UNO 板实物示意图

搭建开发 Arduino 的硬件环境非常简单，只需要一根 USB 线将 Arduino 板与计算机连接即可。这里 USB 线既可以给 Arduino 供电，也可以将程序下载到 Arduino 板中。

5.2.2 开发环境安装与配置

Arduino 板有免费配套的开发环境，可以在官网（www. arduino. cc）上进行下载，有 Windows、Mac OS、Linux 版本，读者可根据自己的需要选择。安装完成后，在安装目录下双击 Arduino. exe 图标，如图 5-2 所示。

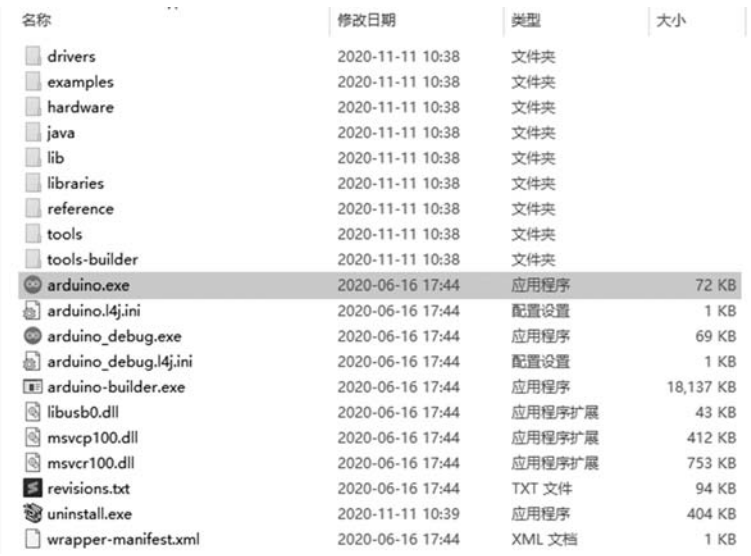


图 5-2 双击 Arduino 图标

因为 Arduino 包含不同功能的硬件，必须选择正确的 Arduino 板名称，单击菜单栏“工具”→“开发板："Arduino Uno"”→Arduino Uno 命令，如图 5-3 所示。

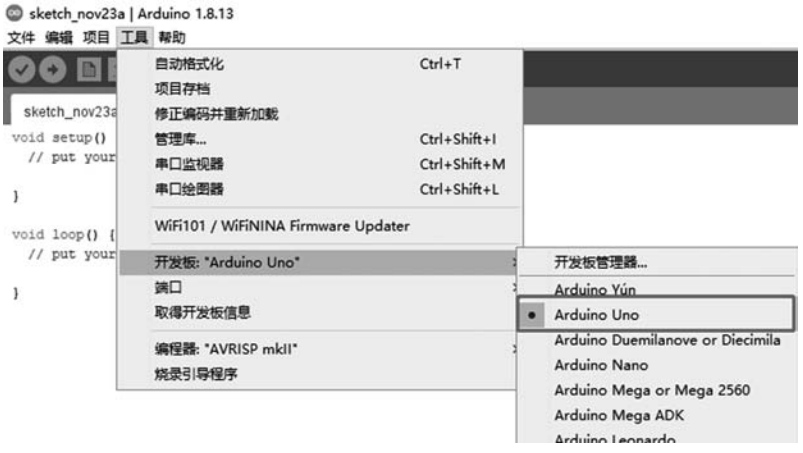


图 5-3 选择正确的 Arduino 板

在计算机端使用串口来将程序下载到 Arduino 板中，使用 USB 连接计算机和 Arduino

板后需要选择正确串口号，单击“工具”→“端口”命令，选择正确的 COM 端口，如图 5-4 所示。

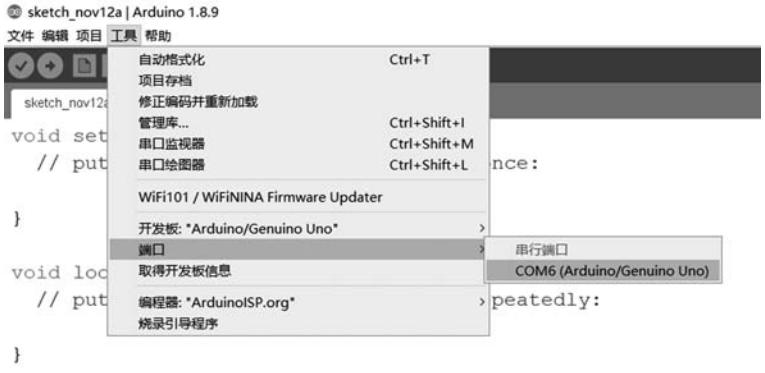


图 5-4　选择串口端口号

在默认的编程环境中，包含很多 Arduino 的例程，单击“文件”→“示例”→01. Basics→Blink 命令，打开 LED 闪烁的例程，如图 5-5 所示。

图 5-5　打开 LED 闪烁的例程

接下来需要将程序下载到 Arduino 开发板中，在开发环境有调试程序的快捷键，如图 5-6 所示。

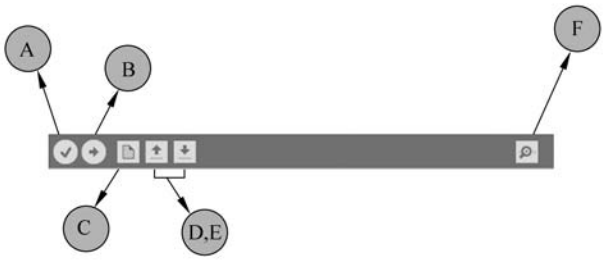


图 5-6　调试程序快捷按钮

其中：

A 为编译按钮，用于检查是否存在任何编译错误；

B 为程序下载按钮，用于将程序上传到 Arduino 板；

C 用于创建新程序文件的快捷方式；

D 用于直接打开示例文件之一；

E 用于保存文件；

F 用于与 Arduino 进行串口通信。

单击 A 按钮编译完成后，单击 B 按钮将程序下载到 Arduino 板中，如果上传成功，则状态栏中将显示“上传成功。”的消息，同时读者可看到板上的 LED 开始闪烁。

5.2.3　Arduino 程序运行与调试方法

学习先从模仿开始，可先从成功的例子中了解 Arduino 的编程思想。Blink 工程的代码如下。

```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
*/

//Pin 13 has an LED connected on most Arduino boards.
//give it a name:
int led = 13;

//the setup routine runs once when you press reset:
void setup() {
  //initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

//the loop routine runs over and over again forever:
void loop() {
```

```
  digitalWrite(led, HIGH);          //turn the LED on (HIGH is the voltage level)
  delay(1000);                      //wait for a second
  digitalWrite(led, LOW);           //turn the LED off by making the voltage LOW
  delay(1000);                      //wait for a second
}
```

从软件结构上看，主要包含 3 部分：变量声明部分、初始化部分以及主要逻辑部分。

变量申明部分一般位于代码最上方，因为在 C/C++语言中遵循先申明后使用的原则，通常在这部分申明所有需要使用的变量。

初始化部分位于 setup()函数中。setup()函数是程序运行的最开始执行的部分，通常在此函数内初始化变量、引脚模式、启动库函数等等，这个函数在程序运行的生命周期内只会执行一次。

主要逻辑部分位于 loop()函数中。当 setup()函数执行完毕时，就会执行 loop()函数。从实际代码中看，在函数内部并没有任何死循环的操作，但是灯会不断闪烁。即可以把 loop()函数看成一个死循环的函数，此函数会不停重复运行。在 loop()函数体内，digitalWrite(led，HIGH)函数是将对 led 引脚电平置高，delay()函数是 Arduino 自带的延时函数，延时最小单位为 1ms，delay(1000)即延时 1s。

Arduino 是通过串口下载程序的，在程序运行过程中，也可以利用此串口来打印调试信息，从而快速确定程序执行错误的地方。使用方法也非常简单，使用 Serial. begin()函数初始化串口，传递参数为比特率。可以通过 Serial. print()函数将调试信息通过串口发送给计算机。以下代码将 LED 状态进行打印。

```
int led = 13;

//the setup routine runs once when you press reset
void setup() {
  //initialize the digital pin as an output
  pinMode(led, OUTPUT);
  //初始化串口
  Serial.begin(9600);
}

//the loop routine runs over and over again forever
void loop() {
  digitalWrite(led, HIGH);          //turn the LED on (HIGH is the voltage level)
  Serial.println("led has turned on.");
  delay(1000);                      //wait for a second
  digitalWrite(led, LOW);           //turn the LED off by making the voltage LOW
  Serial.println("led has turned off.");
  delay(1000);                      //wait for a second
}
```

打开串口监视窗口，可以看到 Arduino 打印的调试信息，如图 5-7 所示，并将监视器右下角的波特率选择为 Serial. begin()传入的比特率。

串口通信是相互的，也可以使用计算机发送控制指令给 Arduino，从而完成计算机对

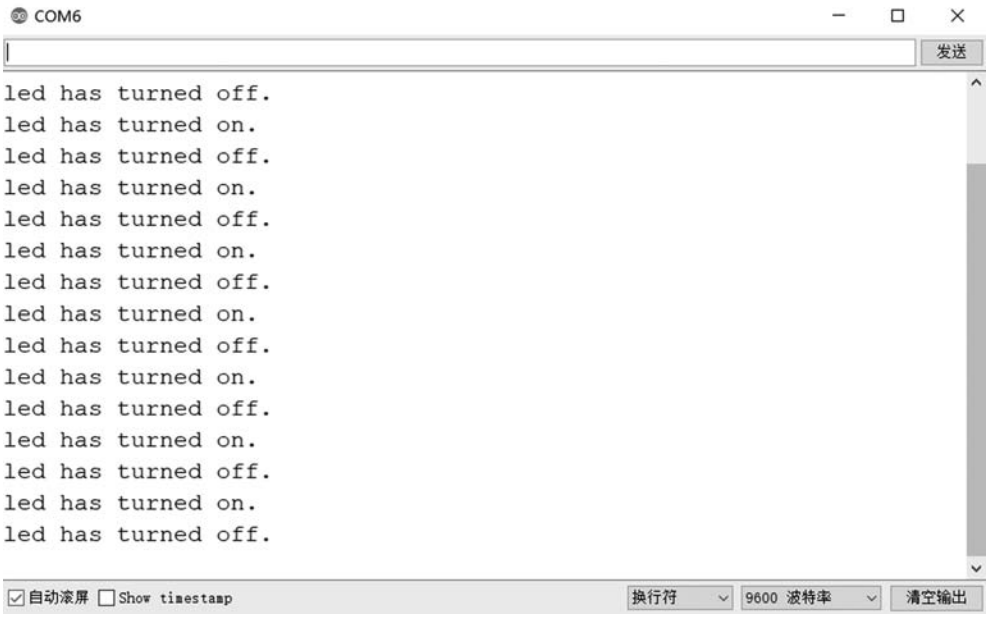


图 5-7　查看调试信息

Arduino 的控制。在串口监视器中的发送文本框中输入要发送的字符，单击“发送”按钮即可将数据发送给 Arduino。在 Arduino 中可以通过 Serial.available()函数获取串口接收到的字符个数，Serial.read()函数获得对应字符。下面为一个接收函数的例子。

```
String recData = "";                          //声明字符串变量,在 C 语言中没有字符串定义

void setup()
{
  Serial.begin(9600);                         //设定的比特率
}

void loop()
{
   while (Serial.available() > 0)             //判断是否有可用数据
    {
        recData += char(Serial.read());       //读数据
        delay(2);
    }

   if (recData.length() > 0)
    {
       Serial.println(recData);
       recData = "";
    }
}
```

5.2.4　Arduino 加载其他库函数

在 Arduino 的开发软件中默认自带一些常用的驱动库函数，但对于复杂的应用是不够用的，因此很多时候需要下载并使用第三方的驱动库。

如图 5-8 所示，在 Arduino 开发软件的菜单栏中单击“项目”→“加载库”→“管理库”命令，打开“库管理器”对话框。

图 5-8　打开管理器命令

在文本框中输入需要下载的驱动库名称，单击“安装”按钮，即可安装成功，如图 5-9 所示。

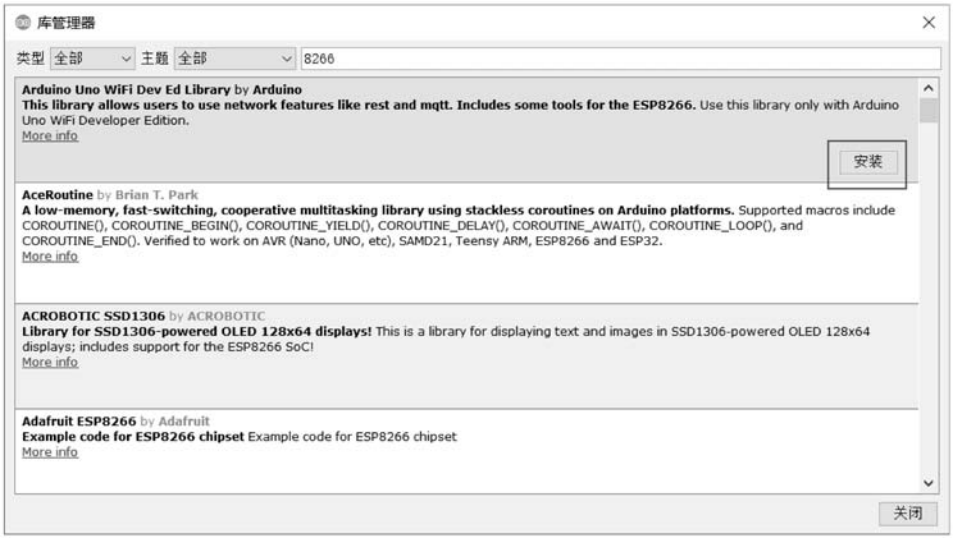


图 5-9　安装驱动库

安装好的驱动库，会出现在“项目”→“加载库”→“推荐的库”中，单击选中即可，最后在代码区域添加头文件(#include < xxx >)便可调用该库。

在后面章节中如发现开发软件中没有对应的驱动库，可使用该方法进行添加。

5.3 模拟与数字的桥梁

模拟电路中充满着各种波形，在时间和数值上均具有连续性，而数字电路中时间和数值具有离散性，通过 0 与 1 两个数值的组合来表示各种信息。它们之间可以通过 ADC 与 DAC 进行转换。

5.3.1 ADC

ADC 为 Analog to Digital Converter 的缩写，指模数转换器，用于将模拟信号转换成数字信号。在 Arduino UNO 板中支持 5 个 ADC 输入端口，为 A0～A5，如图 5-10 所示。其分辨率为 10 位，默认以输入电压作为基准电压($V_{ref}=5.0V$)。即 0～5V 的电压经过 ADC 转换后的变成 0～1023($2^{10}-1$)的数值。

显然，通过 ADC 的值可以推出真实的模拟电压值，公式为

$$V = \text{ADC\_Value} \times (V_{ref}/1024) \tag{5-1}$$

下面对 ADC 的使用进行举例说明。如图 5-11 所示，使用滑动变阻器来对 Arduino 输入电压进行分压，将滑动变阻器一侧接到板上的 GND，另外一侧引脚接到 5V 接口，将滑动变阻器中间引脚连接到板子上的模拟输入引脚 A0 上。

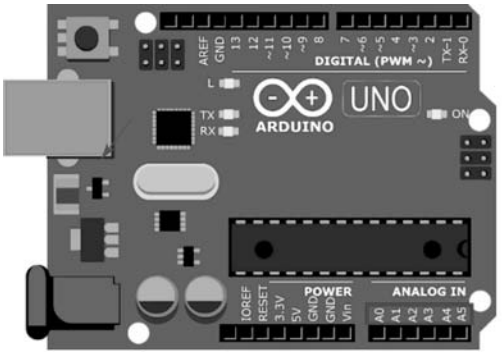


图 5-10 Arduino ADC 接口

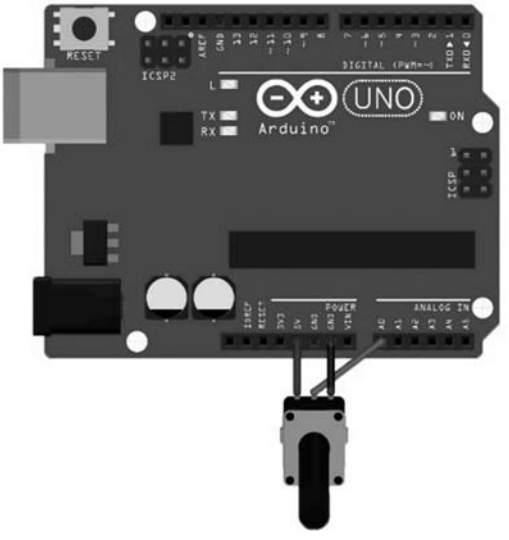


图 5-11 ADC 测试电路图

可以通过 analogRead()函数读取模拟值，完整代码如下：

```
void setup() {
   Serial.begin(9600);                    //使用 9600bps 的比特率进行串口通信
```

```
}

void loop() {
  int n = analogRead(A0);                    //读取 A0 口的电压值
  double vol = n * (5 / 1024.0) * 100;       //读取模拟值,结果是乘以 100 后的值
  Serial.println(vol);                       //串口输出模拟值
  delay(500);                                //等待 0.5s,控制刷新速度
}
```

打开串口调试窗口可以看到电压值在变化,在图形绘制窗口看到波形,如图 5-12 所示。

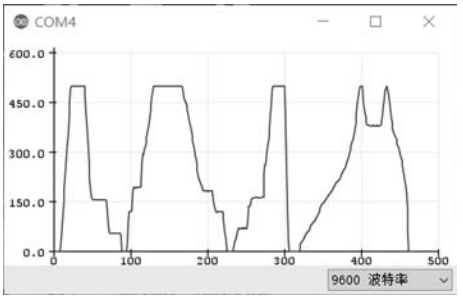


图 5-12　图形绘制窗口显示 ADC 值

可以看出,analogRead()函数读取的值与其参考电压 V_{ref} 有关,在 Arduino 中可以通过 analogReference(type)函数进行配置,传入的 type 值有 5 个:

(1) DEFAULT——默认模式,为 Arduino 的输入电压;

(2) INTERNAL——内置参考值;

(3) INTERNAL1V1——使用内置 1.1V 参考电压;

(4) INTERNAL2V56——使用内置 2.56V 参考电压;

(5) EXTERNAL——使用外部 AREF 引脚电压作为参考电压,注意请勿使用小于 0V 或大于 5V 的任何值,否则可能会损坏 Arduino 板。

5.3.2　DAC

DAC 为 Digital to Analog Converter 的缩写,即数模转换器,与 ADC 正好相反,DAC 可以将数字信号转换成模拟信号。DAC 接口在 Arduino UNO 的接口位置如图 5-13 所示,共支持 6 个 DAC 输出接口(第 3、5、6、9、10、11 引脚)。

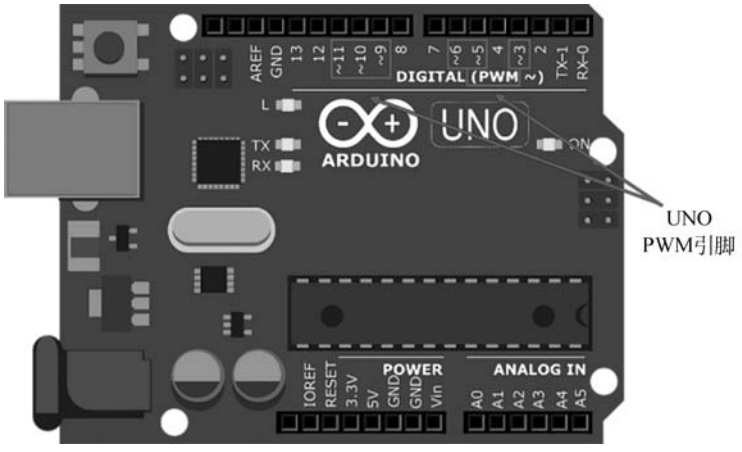


图 5-13　Arduino DAC 输出接口

在 Arduino 中 DAC 比较特殊，它产生的模拟值并不是稳定的一个电压值，而是以 PWM 波的形式输出，也可以将 Arduino 的 DAC 输出理解为有效值输出。可以通过 analogWrite(pin,value)函数输出 PWM。传递参数 pin 为 DAC 输出引脚；传递的参数 value 为占空比，取值范围为 0～255，对应真实 PWM 的占空比为 0%～100%。

下面举一个简单的例子(呼吸灯)介绍 Arduino 中 PWM 的输出，原理图如图 5-14 所示。

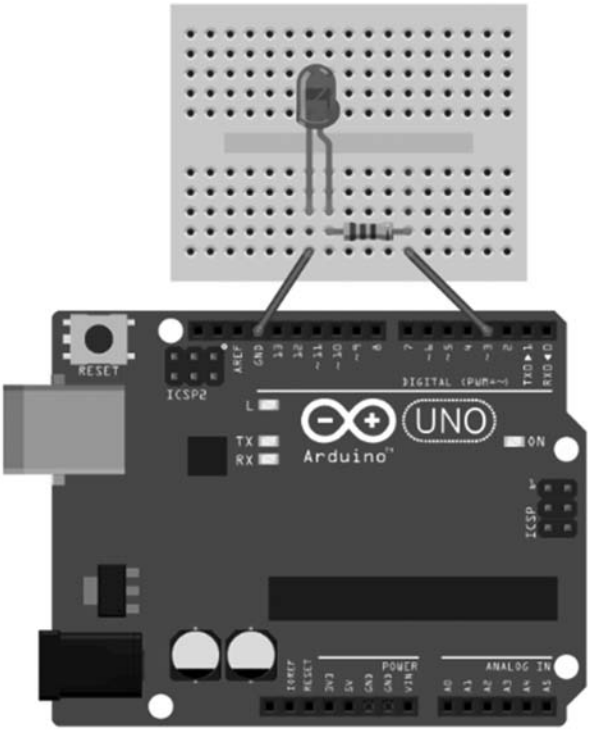


图 5-14　呼吸灯电路图

在 Arduino Uno 中，引脚 5 和引脚 6 的 PWM 输出频率约为 980Hz，其余引脚为 490Hz。但人眼识别连贯图像速度约为 24 帧/秒，因此当 Arduino 输出的 PWM 波形加载到 LED 上时，人眼无法看到 LED 的闪烁，而是看到 LED 的亮暗程度。那么当 PWM 输出不同占空比时，LED 的亮度会随之变化。

LED 使用从灭到亮的状态模拟人的吸气，使用从亮到灭的状态模拟人的呼气，在 Arduino 上对应是 PWM 的占空比的变化。完整代码如下所示。

```
int ledPin = 3;

void setup()
{
  pinMode(ledPin,OUTPUT);
}
```

```
void loop()
{
  for (int a = 0; a <= 255;a++)          //循环语句,控制 PWM 亮度的增加
  {
    analogWrite(ledPin,a);
    delay(8);                            //当前亮度级别维持的时间,单位为毫秒
  }
  for (int a = 255; a >= 0;a-- )         //循环语句,控制 PWM 亮度减小
  {
    analogWrite(ledPin,a);
    delay(8);                            //当前亮度级别的维持时间,单位为毫秒
  }
}
```

PWM 在实际应用中十分广泛,除了在第 4 章中介绍 PWM 在开关电源中的应用,其在电机控制领域也得到了广泛应用,本章后面将详细介绍。

5.4　人机接口

电子产品最终的服务对象是人,因此人机接口就显得尤为重要,它可以让人与电子系统之间建立联系并交互信息。前文通过串口打印调试信息让开发者了解程序的运行流程,这也可以看成是人机接口的一种方式。

人机接口分为输入接口与输出接口。输入接口是人对机器的控制,在电子系统中通常采用按键的方式进行输入。输出接口则是电子系统对人的反馈,在电子系统中通常采用 LCD、OLED 等显示屏等对必要信息进行显示。

5.4.1　按键输入

在对 LED 灯控制的时候将引脚配置为了输出模式,当需要读取按键状态时候需要将这个引脚配置为输入模式。如图 5-15 为测试按键输入原理图,通过两个微动开关分别控制 LED 的亮与灭。

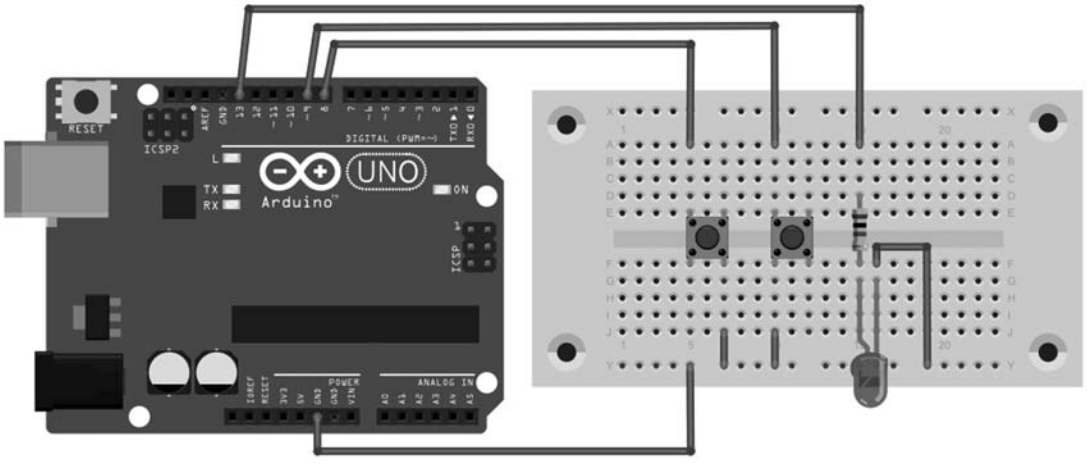


图 5-15　按键输入原理图

需要在 setup()函数中配置 led(13)、开按键(9)、关按键(8)的初始化，并在 loop()函数中判定开关状态从而改变 led 的状态，完整代码如下所示。

```
void setup()
{
  pinMode(13, OUTPUT);
  pinMode(9, INPUT_PULLUP);                //按键设置为 input 的状态
  pinMode(8, INPUT_PULLUP);                //按键设置为 input 的状态
}

void loop()
{
  if (digitalRead(buttonApin) == LOW)      //按键在按下后是 LOW 的状态
  {
    digitalWrite(ledPin, HIGH);
  }
  if (digitalRead(buttonBpin) == LOW)      //按键在按下后是 LOW 的状态
  {
    digitalWrite(ledPin, LOW);
  }
}
```

这种方式的按键输入开发非常方便，但当需要的按键很多时，这种方式就会大量占用硬件的输入引脚，导致引脚不够用。下面介绍解决这种多按键情况的两种方式。

5.4.2 矩阵式 4×4 键盘输入

矩阵式键盘又称为行列式键盘，它是用 4 条 I/O 线作为行线，4 条 I/O 线作为列线组成的键盘，如图 5-16 为其实物与原理示意图。在行线和列线的每一个交叉点上设置一个按键。这样键盘中按键的个数是 4×4 个，这种行列式键盘结构能够有效地提高单片机系统中 I/O 口的利用率。

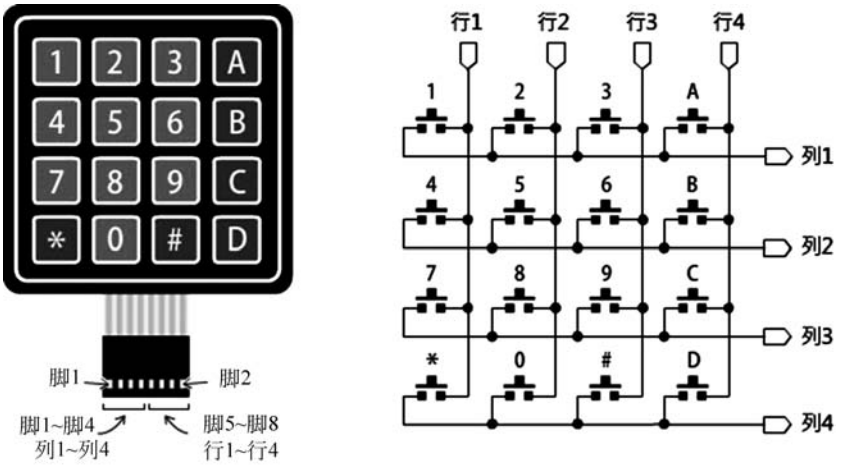


图 5-16 矩阵式按键的实物与原理示意图

判定的一般方法为：分别在行 1～行 4 输出低电平时读取列 1～列 4 的状态，再使列 1～列 4 输出低电平，读取行 1～行 4 的状态。将两次读取结果组合起来就可以得到当前按键的特征编码。

如图 5-17 所示为矩阵式按键与 Arduino 连接的硬件图。

在 Arduino 中有封装好的矩阵式按键的库函数（Keypad. h），定义好键盘行数（KEY_ROWS）、键盘列数（KEY_COLS）、依照行与列排序的矩阵式按键上的符号（keymap）以及行与列连接的引脚（rowPins 与 colPins）的相关变量，调用 Keypad()函数并传递相关参数完成矩阵键盘初始化。最后调用 getKey()方法来获取键盘接口。完整代码如下所示。

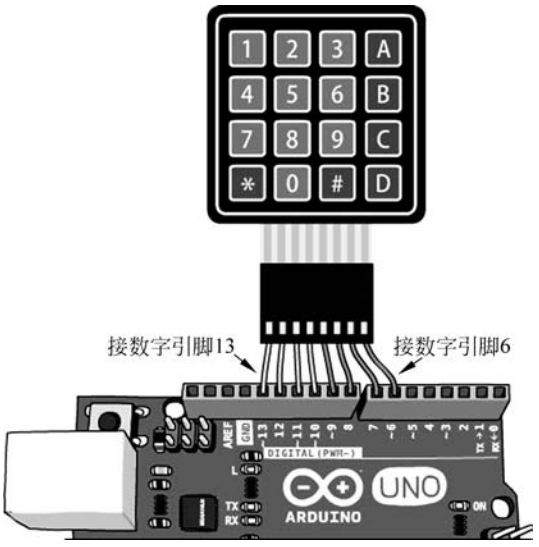


图 5-17　矩阵式按键与 Arduino 连接的硬件图

```
#include <Keypad.h>                                    //调用 Keypad 程序库

#define KEY_ROWS 4                                     //按键模块的行数
#define KEY_COLS 4                                     //按键模块的列数

//依照行、列排序的矩阵式按键上的符号
char keymap[KEY_ROWS][KEY_COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte colPins[KEY_COLS] = {9, 8, 7, 6};                 //按键模块，列 1～列 4 的引脚
byte rowPins[KEY_ROWS] = {13, 12, 11, 10};             //按键模块，行 1～行 4 的引脚

//初始化矩阵式键盘
Keypad myKeypad = Keypad(makeKeymap(keymap), rowPins, colPins, KEY_ROWS, KEY_COLS);

void setup(){
  Serial.begin(9600);
}

void loop(){
  //通过 Keypad 中的 getKey()方法读取按键
  char key = myKeypad.getKey();

  if (key){                                            //若有按键被按下
```

```
        Serial.println(key);              //打印被按下的按键
    }
}
```

5.4.3 AD 采样键盘输入

矩阵式按键输入相比于单个按键输入节约了很多硬件引脚资源，但在引脚特别稀缺的场景中仍然不适用。下面介绍 AD 采样键盘电路，只需要一个引脚即可判定多个按键的输入状态，如图 5-18 所示为对应的原理图。

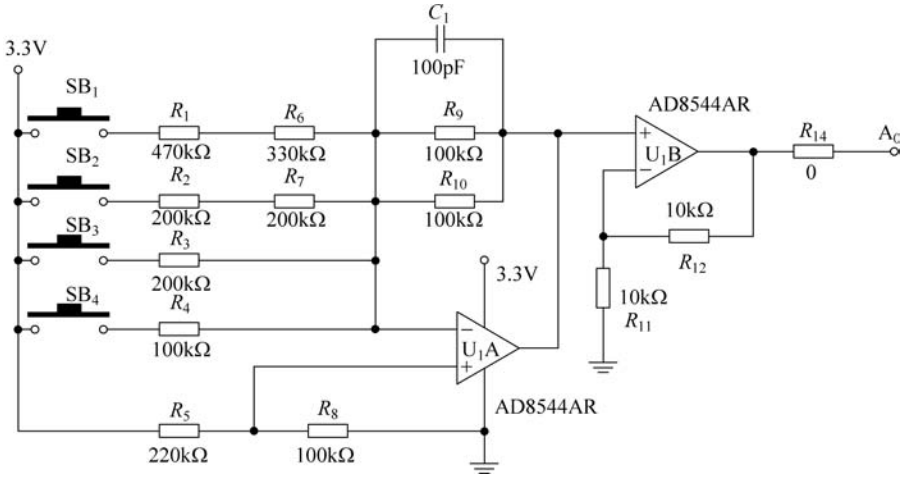


图 5-18 AD 采样键盘电路

当 SB_1～SB_4 都没有被按下时，运算放大器 U_1A 与周围电阻组成的是电压跟随电路，R_5 与 R_8 分压将电源电压缩小大约 3 倍作为电压跟随器的输入，再经过 U_1B 与周围电阻组成的放大电路(放大倍数为 3)得到最终输出电压，约为 3.3V，即没有按键被按下的时候输出电压约 3.3V。当 SB_1～SB_4 有按键被按下时，U_1A 与周围电阻组成的是减法电路，根据与微动开关串联电阻的阻值不同，减去的电压也不同，从而运算放大器最终输出的电压不同，因此可以根据运算放大器输出的电压来判定哪个按键被按下。

5.4.4 LCD1602 显示

LCD1602 是一种常用的工业字符型液晶显示器，能够同时显示 16×02 即 32 个字符，实物图如图 5-19 所示。

图 5-19 LCD1602 实物

LCD1602 的引脚说明如表 5-1 所示。

表 5-1　LCD1602 引脚说明

引　脚	符　号	说　明
1	GND	接地
2	VCC	5V 正极
3	V0	对比度调整，接正极时对比度最弱
4	RS	寄存器选择，1 数据寄存器(DR)，0 指令寄存器(IR)
5	R/W	读写选择，1 读，0 写
6	EN	使能端，高电平读取信息，负跳变时执行指令
7～14	D0～D7	8 位双向数据
15	BLA	背光正极
16	BLK	背光负极

LCD1602 与 Arduino 连接引脚对照表如表 5-2 所示，电路连接示意图如图 5-20 所示。

表 5-2　LCD1602 与 Arduino 连接引脚对照表

LCD1602		Arduino UNO
GND	→	GND
VCC	→	5V
V0	→	旋转变阻器可调引脚
RS	→	3 引脚
R/W	→	GND
EN	→	5 引脚
D0～D3	→	—
D4～D7	→	10～13 引脚
BLA	→	5V
BLK	→	5V

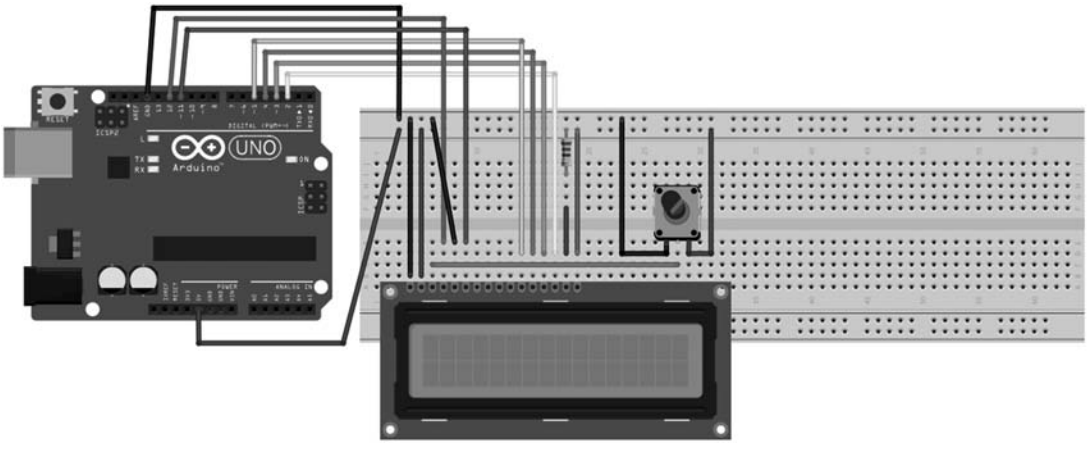


图 5-20　Arduino 与 LCD1602 连接示意图

在 Arduino 中可使用 LiquidCrystal 驱动库对 LCD1602 进行开发，代码如下所示。

```
//引入依赖
#include <LiquidCrystal.h>

//初始化针脚
const int rs = 3, en = 5, d4 = 10, d5 = 11, d6 = 12, d7 = 13;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);

void setup() {
    //设置 LCD 要显示的列数、行数,即 2 行 16 列
    lcd.begin(16, 2);
    //输出 Hello World
    lcd.print("hello, world!");
}

void loop() {
    //设置光标定位到第 0 列,第 1 行(从 0 开始)
    lcd.setCursor(0, 1);
    //打印从重置后的秒数
    lcd.print(millis()/1000);
}
```

5.4.5 OLED 显示

用 LCD1602 进行开发相对比较简单,但最多只能显示 32 个字符而且体积相对较大,下面介绍一种体积小的显示器——OLED。它是利用有机电自发光二极管制成的显示屏,不需背光源,具有对比度高、厚度薄、视角广、反应速度快、使用温度范围广、构造及制程较简单等优异特性。OLED 显示屏可以显示汉字、字符和图案等,智能手环和智能手表等智能设备一般都选择 OLED 显示屏来作为显示器。

OLED 内部由 SSD1306 芯片对界面进行驱动,支持 SPI 与 I^2C 两种通信协议对 OLED 进行驱动,本书主要介绍 I^2C 驱动的 OLED 显示屏,其实物如图 5-21 所示。在 OLED 的坐标系统中左上角是原点,向右是 X 轴,向下是 Y 轴,可实现 128×64 点阵显示。

使用 Arduino 驱动 OLED 原理图如图 5-22 所示。

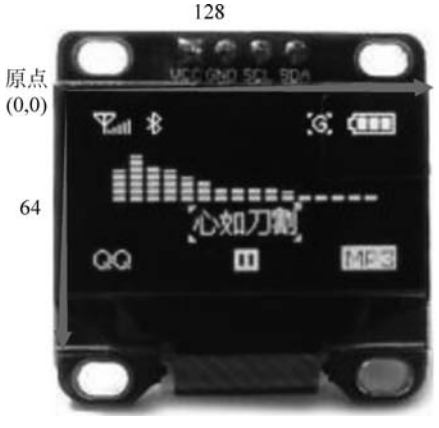


图 5-21 OLED 显示

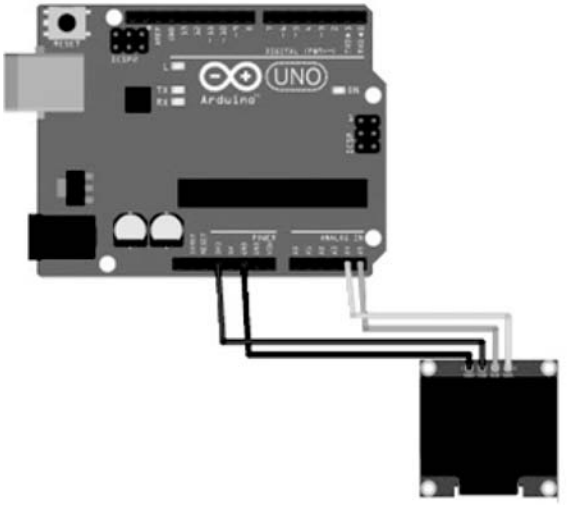


图 5-22 Arduino 驱动 OLED

在 Arduino 中，使用 Adafruit_SSD1306 与 Adafruit_GFX 相结合的形式驱动 OLED。Adafruit_SSD1306 库函数定义了 SSD1306 芯片相关的驱动，Adafruit_GFX 库函数中定义了一系列的绘画方法，包括线、圆、矩形等。

安装完驱动库之后，需要修改 Adafruit_SSD1306 驱动库的配置，默认 SSD1306 驱动的屏的大小是 128×32px，需要修改为 128×64px。进入 Arduino 安装文件夹后，在 libraries→Adfruit_SSD1306-master 文件夹中找到 Adafruit_SSD1306.h 文件，注释代码“#define SSD1306_128_32”，并对“#define SSD_128_64”取消注释，如图 5-23 所示。

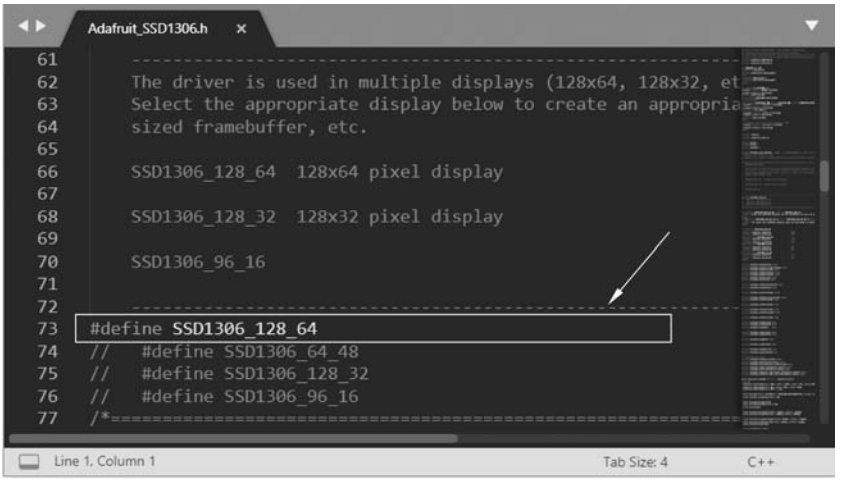


图 5-23　配置 OLED 显示屏像素为 128×64

修改完配置后，需定义 OLED 驱动库对应的头文件以及调用初始化函数，代码如下所示。

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define OLED_RESET 4
Adafruit_SSD1306 display(OLED_RESET);

void setup() {
  Serial.begin(115200);
  delay(500);
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);      //对于 128 * 64 的 I²C 地址为 0x3C

}
```

在初始化函数中需注意在 display.begin()函数中传递正确的 OLED 地址。初始化完成后即可利用 Adafruit_GFX 库函数在 OLED 显示屏中进行显示。这里介绍几个重要的显示函数，如表 5-3 所示，更多驱动函数可以查看其源码。

表 5-3 OLED 驱动库常用显示函数

函 数 名	函数功能
fillScreen()	全屏显示某一颜色,通常用来检测显示屏中是否有坏点
clearDisplay()	清屏操作
display()	数据显示到屏幕上,任意一个绘制操作后都需要调用此函数
drawPixel()	绘制点,传入点的坐标与颜色
drawLine()	绘制线段,传递线段起始坐标与颜色
drawRect()	绘制空心矩形,传入矩形左上角坐标、矩形的宽度和高度以及颜色
fillRect()	绘制实心矩形,传入矩形左上角坐标、矩形的宽度和高度以及颜色
drawCircle()	绘制画空心圆,传入圆心坐标、半径以及颜色
fillCircle()	绘制画实心圆,传入圆心坐标、半径以及颜色
setTextSize()	设置文字大小
setTextColor()	设置文字颜色
print()	打印字符串
println()	打印变量
drawBitmap()	画任意图形,传入左上角坐标、图形数据、图形高度与宽度以及颜色

下面的代码对常用函数进行实现。

```
//显示一个心形的数据
static const uint8_t PROGMEM Heart_16x16[] = {
  0x00,0x00,0x18,0x18,0x3C,0x3C,0x7E,0x7E,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
  0xFF,0xFF,0x7F,0xFE,0x3F,0xFC,0x1F,0xF8,0x0F,0xF0,0x07,0xE0,0x03,0xC0,0x00,0x00
                                    //未命名文件 0
};

void loop() {
  test_SSD1306();
}

void test_SSD1306(void){
  //实例 1.检测全屏显示(看看有没有大面积坏点)
  display.fillScreen(WHITE);
  display.display();
  delay(2000);

  //实例 2.画点,点坐标(10,10)
  display.clearDisplay();          //clears the screen and buffer
  display.drawPixel(10, 10, WHITE);
  display.display();
  delay(2000);

  //实例 3. 画线,从(0,0)到(50,50)
  display.clearDisplay();          //clears the screen and buffer
  display.drawLine(0, 0,50,50, WHITE);
  display.display();
  delay(2000);
```

```
  //实例 4.画空心矩形,左上角坐标(x0,y0),右下角坐标(x1,y1)
  display.clearDisplay();                  //clears the screen and buffer
  display.drawRect(0,0,128,64,WHITE);
  display.display();
  delay(2000);

  //实例 5.画实心矩形
  display.clearDisplay();                  //clears the screen and buffer
  display.fillRect(0,0,64,64,WHITE);
  display.display();
  delay(2000);

  //实例 6.画空心圆
  display.clearDisplay();                  //clears the screen and buffer
  display.drawCircle(20,20,20,WHITE);
  display.display();
  delay(2000);

  //实例 7.画实心圆
  display.clearDisplay();                  //clears the screen and buffer
  display.fillCircle(20,20,20,WHITE);
  display.display();
  delay(2000);

  //实例 8.画心形
  display.clearDisplay();                  //clears the screen and buffer
  display.drawBitmap(16,16,Heart_16x16,16,16,WHITE);
  display.display();
  delay(2000);

  //实例 9.显示英文数字
  display.clearDisplay();                  //clears the screen and buffer
  display.setTextSize(1);
  display.setTextColor(WHITE);
  display.setCursor(0,0);
  display.println("Hello, Arduino!");
  display.setTextColor(BLACK, WHITE);      //'inverted' text
  display.println(3.141592);
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.print("0x"); display.println(0xDEADBEEF, HEX);
  display.display();
  delay(2000);
}
```

代码中对应实例 1～9 的执行效果如图 5-24 所示。

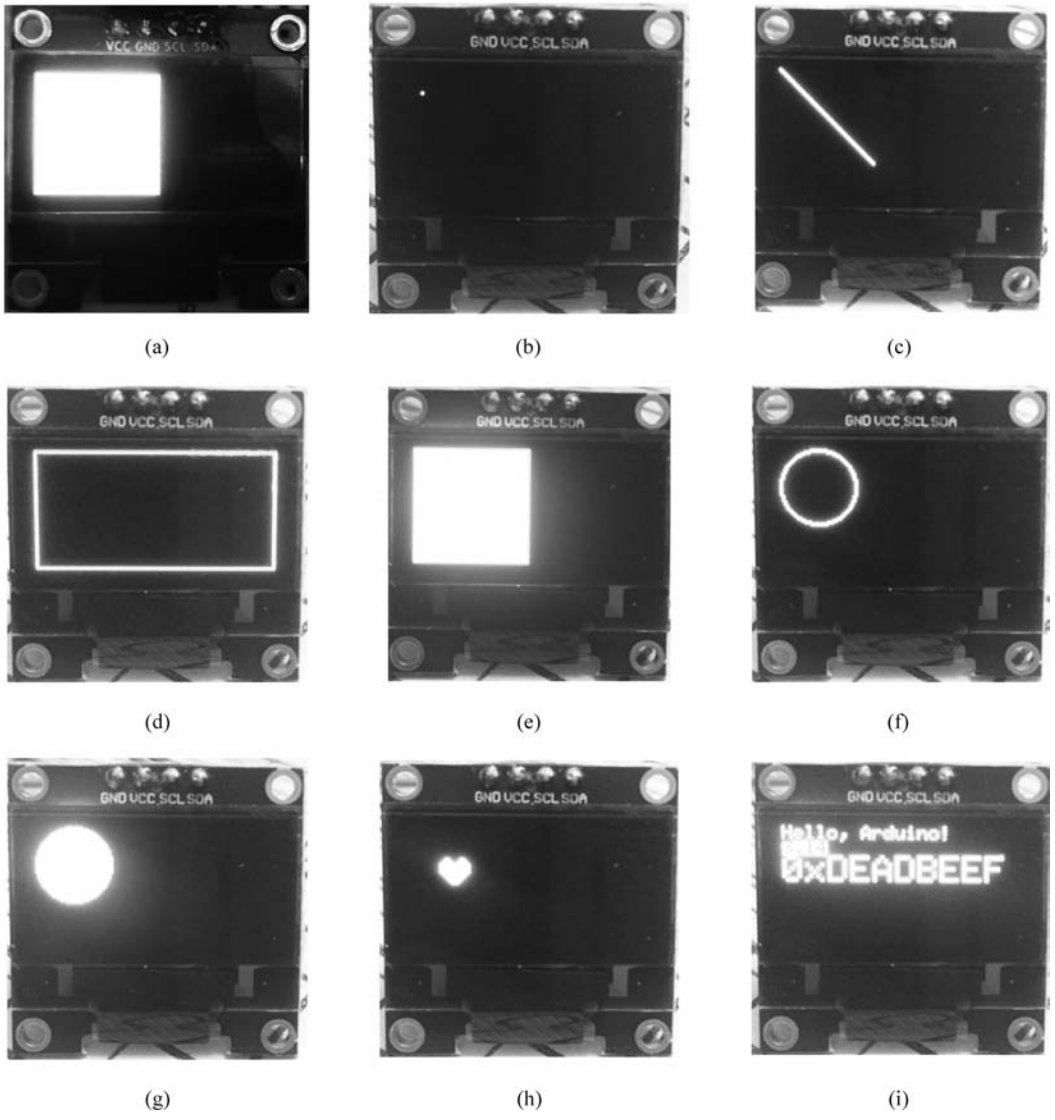


图 5-24　OLED 运行效果图

5.5　常用传感器

人类通过五官感知世界，传感器则是单片机系统的感知设备，涉及控制的领域几乎都会使用传感器。本节对 Arduino 中常用的传感器进行介绍。

5.5.1　空气温湿度传感器

DHT22(也称为 AM2302)是常用数字输出的空气温湿度传感器。它使用电容式湿度传感器和热敏电阻来测量周围空气，并在数据引脚上发送数字信号。实物与引脚定义如图 5-25 所示。DHT22 的电源输入范围为 3～5V。测量湿度范围为 0～100%，精度为 2%～

5%。测量温度范围为－40～80℃，精度为±0.5℃。

DHT22 与 Arduino 连接的示意图如图 5-26 所示，将 DHT22 的 DATA 引脚连接到 Arduino 的 2 号引脚号，V_{CC} 引脚连接到 Arduino 板的 5V 电压，GND 引脚连接到 Arduino 板的接地。

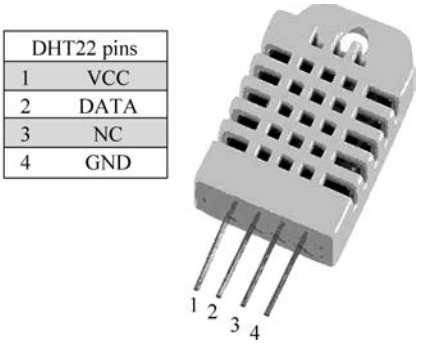


图 5-25　DHT22 实物与引脚定义

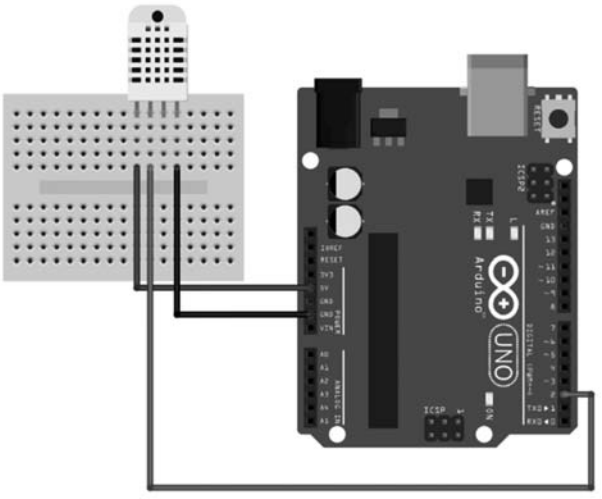


图 5-26　DHT22 与 Arduino 连接的示意图

它们之间的通信采用单总线格式，具体时序图如图 5-27 所示。

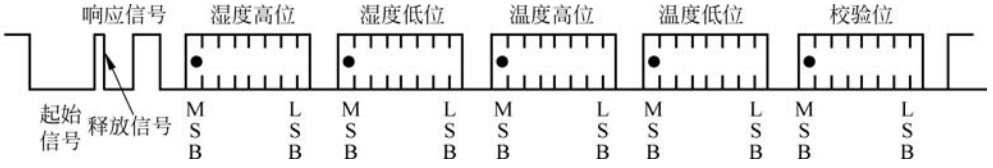


图 5-27　DHT22 单总线协议

可以分为 3 步完成数据读取：

（1）传感器上电阶段。在 DHT22 上电后一般需要等待 2s 以越过传感器的不稳定状态，在此期间不建议向传感器发送任何指令。

（2）DHT22 发送响应。Arduino 控制引脚配置为输出模式，同时配置为低电平，且低电平保持时间不小于 800μs，典型值为 1ms。然后 Arduino 需要将控制引脚配置为输入，释放总线，DHT22 将会发送响应信号，即输出 80μs 的低电平作为应答信号，紧接着 80μs 的高电平通知外设准备接收数据。

（3）接收数据阶段。DHT22 发送完响应后，随着由数据总线 DATA 连续输出 40 位数据，Arduino 可以通过引脚高低电平的变化来获取这 40 位数据。位数据为 0 的格式为：50μs 的低电平加 26～28μs 的高电平；位数据为 1 的格式为：50μs 的低电平加 70μs 的高电平。对应格式信号如图 5-28 所示。

在接收到的 40 位的数据中，第 1～16 位为湿度信息，高字节在前，传感器输出的湿度信息是实际湿度值的 10 倍。第 17～32 位为温度信息，高字节在前，传感器输出的温度值也是

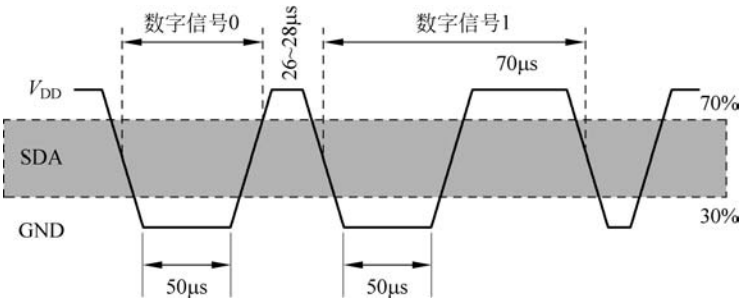


图 5-28　单总线分解时序图

实际值的 10 倍。第 33～40 位为校验位，校验位＝湿度高 8 位＋湿度低 8 位＋温度高 8 位＋温度低 8 位。如图 5-29 所示为 DHT22 传输数据示例。

00000010	10010010	00000001	00001101	10100010
湿度高8位	湿度低8位	温度高8位	温度低8位	校验位

图 5-29　单总线数据示例

校验位：00000010＋10010010＋00000001＋00001101＋10100010＝10100010，数据校验正确；

湿度：00000010＋10010010＝0X0292(十六进制)＝658，即湿度为 65.8%RH；

温度：00000001＋00001101＝0X10D(十六进制)＝269，即温度为 26.9℃。

需要特殊说明的是，当温度低于 0℃时温度数据的最高位为 1。

DHT22 单总线协议看起来比较复杂，但在 Arduino 中开发非常容易，可以通过 DHT.h 库函数对 DHT22 进行操作，具体代码如下：

```
#include "DHT.h"

#define DHTPIN 2                                    //DHT22 连接 Arduino 的引脚
#define DHTTYPE DHT22                               //定义传感器型号
DHT dht(DHTPIN, DHTTYPE);                           //初始化 DHT22

void setup() {
   Serial.begin(9600);
   Serial.println("DHTxx test!");
   dht.begin();
}

void loop() {
   delay(2000);                                     //在测试之前等待几秒钟
   float h = dht.readHumidity();                    //读取温度或湿度大约需要 250ms
   float t = dht.readTemperature();                 //默认读取摄氏度温度
   float f = dht.readTemperature(true);             //读取华氏温度

   if (isnan(h) || isnan(t) || isnan(f)) {          //检查是否有任何读取失败并提前退出
      Serial.println("Failed to read from DHT sensor!");
```

```
        return;
    }

    float hif = dht.computeHeatIndex(f, h);          //计算热指数
    float hic = dht.computeHeatIndex(t, h, false);
    Serial.print("Humidity: ");
    Serial.print(h);
    Serial.print(" %\t");
    Serial.print("Temperature: ");
    Serial.print(t);
    Serial.print(" *C ");
    Serial.print(f);
    Serial.println(" *F");
}
```

结果如图 5-30 所示。

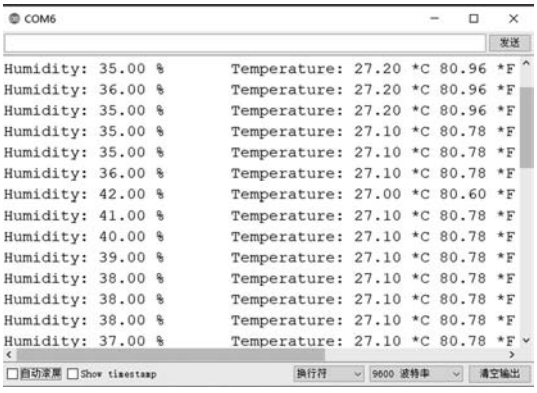


图 5-30　串口显示空气温湿度的结果

5.5.2　超声波传感器

HC-SR04 是常用的超声波传感器，使用声呐来确定物体的距离，能够实现非接触式检测，且具有准确度高、读数稳定、易于使用等优点，其实物如图 5-31 所示。HC-SR04 输入电源电压为 5V，工作电流为 15mA，测距距离范围为 2～400cm，分辨率为 0.3cm，测量角度为 30°。

在 HC-SR04 传感器中配有超声波发射器和接收器模块，被测量材质需要反射超声波效果好，因此在布料等柔软材料上可能误差较大。

传感器有 4 个端子：+5V、Trigger、Echo 和 GND，Arduino 可以通过 Trigger 与 Echo 引脚获取测得距离。具体步骤为：

图 5-31　HC-SR04 传感器实物

(1) 使用 Arduino 控制 HC-SR04 传感器的 TRIG 引脚最少维持 10μs 的高电平信号；

(2) HC-SR04 传感器会自动发送 8 个 40kHz 的方波，传感器自动检测距离信号；

(3) 若有信号返回，则通过 ECHO 输出一段时间的高电平，高电平持续的时间就是超声波从发射到返回的时间。测试距离 $d=$(高电平时间 $t\times$声速 v(340m/s))/2，这里 t 的单位为 s。如果距离 d 的单位为 cm，可以简化公式为 $d=t/(29\times2)$，这里 t 的单位为 ms。

对应时序图如图 5-32 所示。

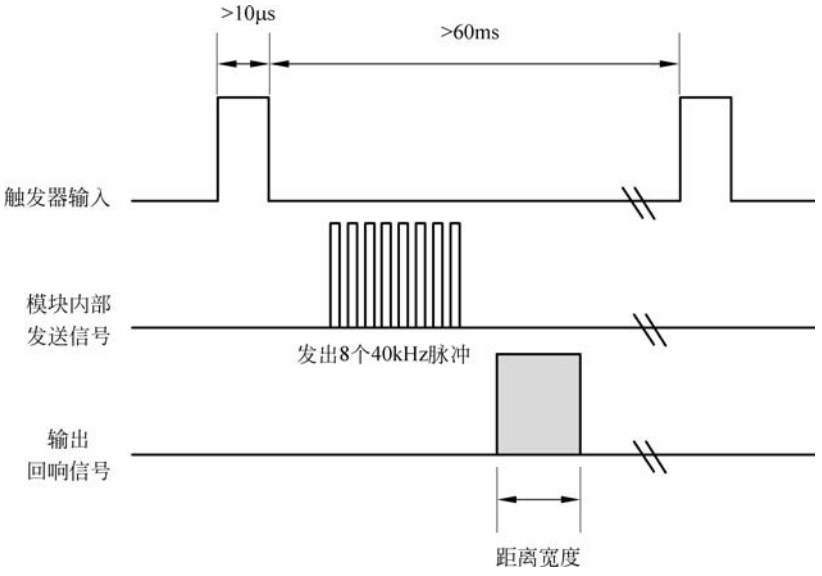


图 5-32　HC-SR04 获取距离时序图

如图 5-33 所示为 Arduino 与 HC-SR04 传感器连接示意图，将传感器的+5V 引脚连接到 Arduino 板上的+5V，传感器的 Trigger 连接到 Arduino 板上的数字引脚 7，将传感器的 Echo 连接到 Arduino 板上的数字引脚 6，将传感器的 GND 连接到 Arduino 上的 GND 引脚。

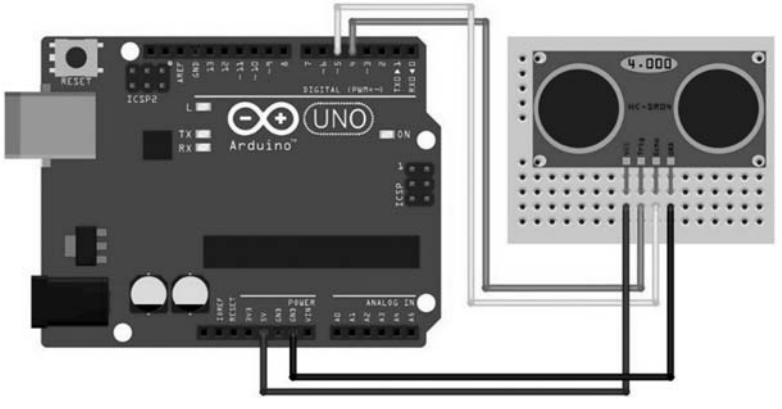


图 5-33　Arduino 与 HC-SR04 传感器连接示意图

可以通过 Arduino 自带 pulseIn()直接获取某一引脚高电平维持的时间，单位为 ms。

实现程序如下：

```
const int pingPin = 7;                                  //Trigger 引脚
const int echoPin = 6;                                  //Echo 引脚

void setup() {
   Serial.begin(9600);                                  //串口调试初始化
}

void loop() {
   long duration, inches, cm;
   pinMode(pingPin, OUTPUT);
   digitalWrite(pingPin, LOW);
   delayMicroseconds(2);
   digitalWrite(pingPin, HIGH);
   delayMicroseconds(10);                               //10μs 高电平触发传感器采集
   digitalWrite(pingPin, LOW);
   pinMode(echoPin, INPUT);
   duration = pulseIn(echoPin, HIGH);                   //计算脉冲维持时间
   cm = microsecondsToCentimeters(duration);            //将脉冲维持时间转换成距离
   Serial.print(cm);
   Serial.print("cm");
   Serial.println();
   delay(100);
}

long microsecondsToCentimeters(long microseconds) {
   return microseconds / 29 / 2;
}
```

打开串口界面，可以看到采集到的距离信息，如图 5-34 所示。

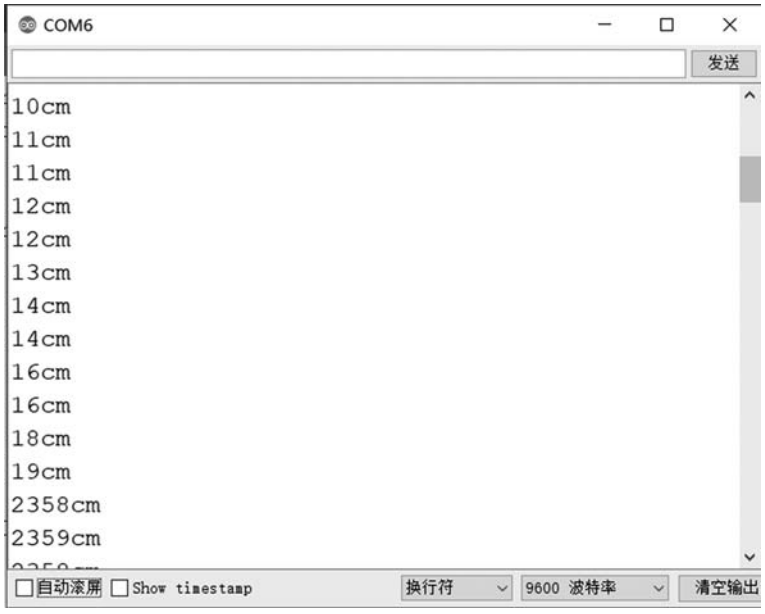


图 5-34　超声波运行效果图

5.5.3 红外传感器

在 Arduino 系统中常用的红外传感器型号为 TCRT5000，该传感器体积小、灵敏度较高，还可以通过转动上面的电位器来调节检测范围，实物如图 5-35 所示。

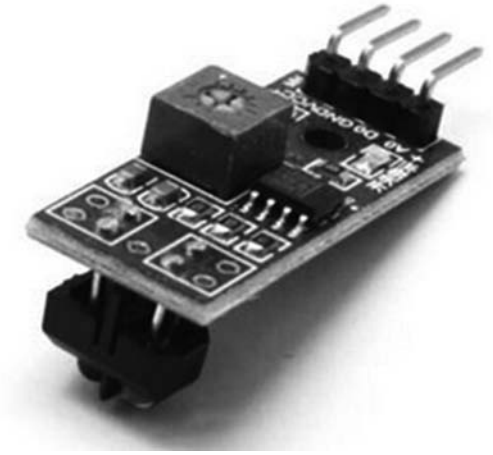


图 5-35 TCRT5000 传感器实物

TCRT5000 包含两个红外二极管，分别为红外发射二极管和红外接收二极管。红外发射二极管不断发射红外线，当发射出的红外线没有被反射回来或被反射回来但强度不够大时，光敏三极管一直处于关断状态，此时模块的输出端为低电平，指示二极管一直处于熄灭状态。当红外线被反射回来且强度足够大，此时模块的输出端为高电平，指示二极管也会被点亮。

此传感器通常也用于循迹小车中。由于黑色具有较强的吸收能力，因此循迹线可为黑色、背景为白色，循迹小车会跟随黑色的轨迹行走。当循迹模块发射的红外线照射到黑线时，红外线将会被黑线吸收，模块会输出低电平，反之输出高电平。使用 Arduino 可以很方便地通过引脚读取此传感器结果，非常简单，此处不再赘述。

5.6 电机控制

电机的发明给人们的生活带来了极大的方便，天上的飞机、地上的汽车、水上的轮船均由电机提供机械动力。在一个自动控制系统中，通常将传感器作为系统的输入，电机作为系统的控制对象，依据传感器获得的环境参数对电机进行精准控制。

直流电机是最常见的电机类型，是指能将直流电能转换成机械能（直流电动机）或将机械能转换成直流电能（直流发电机）的旋转电机。本节将介绍电子设计中常见的直流电动机，包括普通直流电机、伺服电机、步进电机。

5.6.1 普通直流电机

本书中普通直流电机是指内部没有程序控制电路，对外只有两根引线的直流电机。此电机只要通电即可旋转，但其转速、转矩、位置等都需依靠额外的控制电路。如图 5-36 所示为一种普通直流电机实物图。

图 5-36 普通直流电机实物图

1. 开环控制

如果将直流电机直接接入电源，电机可以正常旋转，但是电机的转速等参数是不可调的。因

此为了准确控制电机，需要通过 Arduino＋驱动电路实现。但让电机旋转起来一般需要较大的电流，Arduino 的引脚不能直接控制电机旋转，否则会损坏 Arduino。在电子设计中，一般使用场效应管、三极管或继电器对电机进行驱动。

如图 5-37 所示为 Arduino 通过三极管驱动 5V 直流电机的电路图。使用 Arduino 引脚 4 控制电机的旋转。驱动选用管 IRF730 场效应管，其 $V_{GS(th)}$ 的典型值为 3V，Arduino 可以让其完全导通。因为电动机是感性负载，因此选用 SS2D 二极管作为续流二极管。电阻 R_2 为下拉电阻，防止上电瞬间由于电压不确定引起的电机误动作。

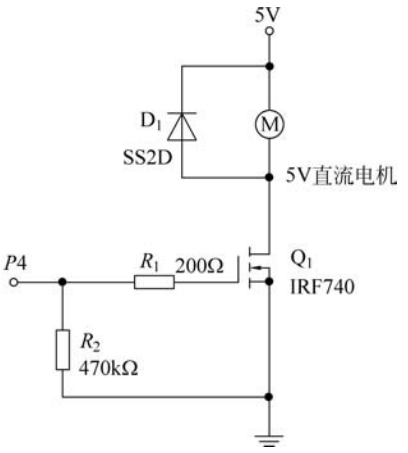


图 5-37　普通电机控制电路

让电动机全速旋转只需要让 $P4$ 为高电平即可。如需控制电机的转速，可使用 Arduino 的 DA 引脚输出 PWM，改变电机两端的有效值，达到改变电动机的转速的目的。这里通过电脑发送 PWM 的占空比给 Arduino，从而实现计算机控制电机转速的目的，代码如下所示：

```
int motorPin = 9;

void setup() {
   pinMode(motorPin, OUTPUT);
   Serial.begin(9600);
   while (! Serial);
   Serial.println("Speed 0 to 255");
}

void loop() {
   if (Serial.available()) {
      int speed = Serial.parseInt();
      if (speed >= 0 && speed <= 255) {
         analogWrite(motorPin, speed);
      }
   }
}
```

串口输出的调试信息如图 5-38 所示。

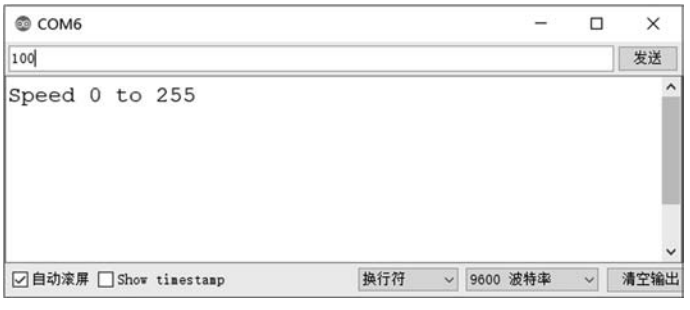


图 5-38　调试电机的串口调试信息

2. PID 闭环控制

事实上，图 5-37 并没有形成一个闭环，因此系统稳定性不够。类比于直流电源变换电路中的 PWM，需要根据输出电压与电流去动态调节 PWM 的占空比，这样才能保证输出电压不随负载的变化而变化。为了将电机转速稳定在设定值 $n_0(t)$，需要不断采集电机的实际转速 $n(t)$。当电机实际转速与设定值之间存在误差时，再根据闭环控制算法动态调节 Arduino 的 PWM 波形，从而稳定电机的转速，其示意图如图 5-39 所示。

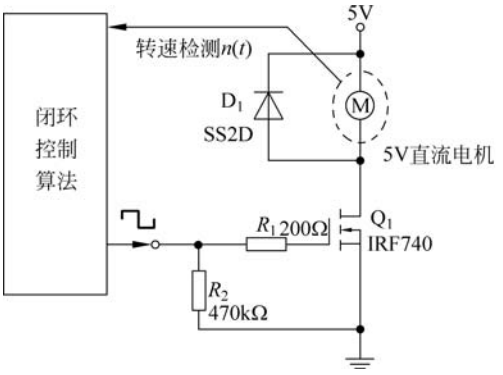


图 5-39　电机闭环控制示意图

其中，闭环控制算法为整个闭环控制的核心，直接影响整个系统的好坏。目前也有很多人对此做出研究，但实际应用中，PID 算法仍然是最为常用的，本书也着重介绍此控制算法。

PID 算法事实上是将偏差比例(Proportion)、积分(Integral)和微分(Differential)通过线性组合构成控制量，用这一控制量对被控对象进行控制的算法。这里的偏差是设定值与采集实际值之间的误差。例如，对于电机闭环控制来说，此偏差为电机实际转速 $n(t)$ 与设定转速 $n_0(t)$ 之间的误差；而对于电源变换来说，此偏差可以为实际输出电压 $u(t)$ 与设定输出电压值 $u_0(t)$ 之间的误差。

常规的 PID 控制算法的框图如图 5-40 所示。其中，$r(t)$ 是给定值，$y(t)$ 是系统的实际输出值，给定值与实际输出值构成控制偏差 $e(t)$，可得

$$e(t)=y(t)-r(t) \tag{5-2}$$

其中，$e(t)$ 作为 PID 控制的输入，$u(t)$ 作为 PID 控制器的输出以及被控对象的输入。所以模拟 PID 算法可以描述为

$$u(t)=K_{\mathrm{p}}\left[e(t)+\frac{1}{T_{\mathrm{i}}}\int_0^t e(t)\mathrm{d}t+T_{\mathrm{d}}\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right] \tag{5-3}$$

其中，K_{p} 为控制器比例系数；T_{i} 为控制器的积分时间；T_{d} 为控制器的微分时间。

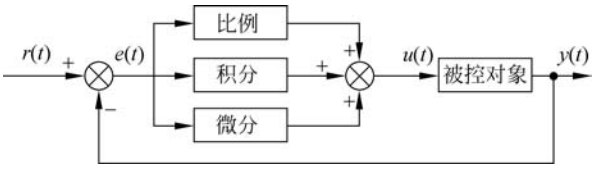


图 5-40　电机调速系统框图

从上式也可看出，PID 算法包括 3 个部分。

(1) 比例部分：数学表达式为 $K_p \times e(t)$。它的作用是对偏差瞬间作出反应。偏差一旦产生控制器立即产生控制作用，使控制量向减少偏差的方向变化。控制作用的强弱取决于比例系数，比例系数越大，控制作用越强，则过渡过程速度越快，控制过程的静态偏差也就越小；但是比例系数越大，也越容易产生振荡，破坏系统的稳定性。故而，比例系数选择必须恰当，才能取得过渡时间少而又稳定的效果。

(2) 积分部分：数学表达式为 $\frac{K_p}{T_i}\int_0^t e(t)\mathrm{d}t$。从此表达式可知，只要系统存在偏差，它的控制作用就不断地增加，因此它最主要的作用是消除系统静态误差。例如，设定转速为 500r/min，如果没有积分部分，可能得到的控制结果为电机稳定在 490r/min 上，存在一个恒定且稳定的误差。另外，积分部分的调节作用虽然可以消除静态误差，但会降低系统的响应速度。

(3) 微分部分：数学表达式为 $K_p T_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$。微分环节的作用是阻止偏差的变化。它是根据偏差的变化趋势(变化速度)进行控制。偏差变化得越快，微分控制器的输出就越大，并能在偏差值变大之前进行修正，具有一定的预判能力。微分作用的引入，将有助于减小超调量，可以加快系统的跟踪速度。但微分的作用对输入信号的噪声很敏感，对那些噪声较大的系统一般不用微分，或在微分起作用之前先对输入信号进行滤波。

显然，比例、积分、微分 3 部分具有不同的功能特点，甚至彼此之间相互制约，因此需要根据实际的应用去合理选择这 3 个参数。

对于式(5-3)来说，积分与微分部分都需要对连续的 $e(t)$进行计算。但在 Arduino 这样的数字电子中，无法得到模拟电子中的连续信号。因此需要对此式中的连续信号进行离散化处理，如式(5-4)所示。

$$u_k = K_p\left[e_k + \frac{T}{T_i}\sum_{j=0}^{k} e_j + \frac{T_d}{T}(e_k - e_{k-1})\right] = K_p e_k + K_i\sum_{j=0}^{k} e_j + K_d(e_k - e_{k-1}) \quad (5\text{-}4)$$

其中，T 为采样周期；k 为采样序列；e_k 为第 k 次采样时刻输入的偏差值；u_k 为第 k 次采样时刻的 PID 控制的输出值；e_{k-1} 为第 $k-1$ 次采样时刻输入的偏差值。K_p、K_i、K_d 分别为比例、积分、微分系数。

只要采样周期足够小，式(5-4)的近似结果就可以足够准确。它是根据模拟 PID 算法直接得到的，其中也包含积分项，需要累加之前所有的偏差。因此被称为**全量式**或**位置式** PID 控制算法。显然，这种算法所需的计算资源较多。

对于计算资源受限的场合，可以使用**增量式** PID 算法，即数字控制器的输出只是控制量的增量 Δu_k。它是根据式(5-4)推导而来的。

由式(5-4)可得到控制器第 $k-1$ 次采样的输出值为

$$u_{k-1} = K_p e_{k-1} + K_i\sum_{j=0}^{k-1} e_j + K_d(e_{k-1} - e_{k-2}) \quad (5\text{-}5)$$

则有

$$\Delta u_k = u_k - u_{k-1} = K_p\left[e_k - e_{k-1} + \frac{T}{T_i}e_k + \frac{T_d}{T}(e_k - 2e_{k-1} + e_{k-2})\right]$$

$$= K_p\left(1 + \frac{T}{T_i} + \frac{T_d}{T}\right)e_k - K_p\left(1 + \frac{2T_d}{T}\right)e_{k-1} + K_p\frac{T_d}{T}e_{k-2}$$

$$= Ae_k + Be_{k-1} + Ce_{k-2} \tag{5-6}$$

显然，如果控制系统采用恒定的采样周期 T，一旦确定了式(5-6)中的 A、B、C，只要前后 3 次测量的偏差即可。则当前输出值应为

$$u_k = \Delta u_k + u_{k-1} \tag{5-7}$$

增量式 PID 与全量式 PID 算法在数字电子控制中都被广泛应用，在本书中都会进行举例。在本章的循迹小车实验中会介绍全量式 PID 控制算法的使用；在第 6 章的数字电源实验中则会介绍增量式 PID 控制算法的使用。

5.6.2 伺服电机

伺服电机事实上本身包含一套内置的控制系统，即无须额外的检测、控制电路与算法即可实现速度、位置精准控制。对于使用者而言，只需要给伺服电机发送命令，它会迅速、准确地执行命令。在电子设计中常用舵机为一种微型伺服电机，其实物与引线定义如图 5-41 所示。

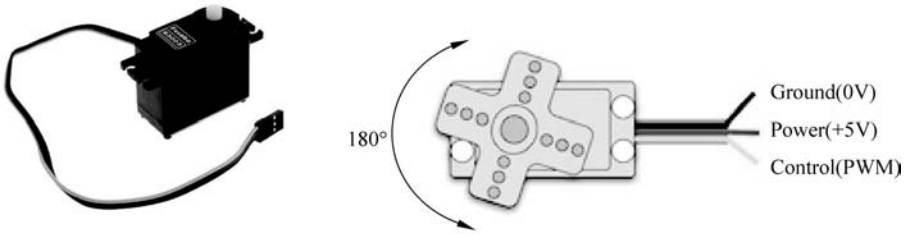


图 5-41 舵机实物图

对舵机控制只需要一根信号线，使用 PPM(脉冲比例调制)信号控制，这里的 PPM 即为伺服电机的控制命令，使得舵机角度和脉冲宽度有关。一般而言，脉宽分布应该为 1～2ms，对应舵机转角为 0°～180°，示意图如图 5-42 所示。

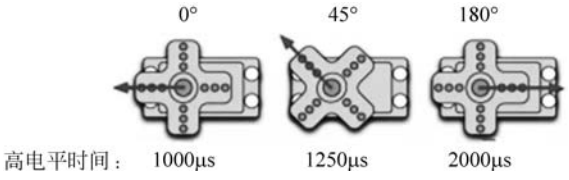


图 5-42 旋转示意图

如图 5-43 所示为 Arduino 控制舵机的原理图，在此图中旨在通过电位器控制舵机的转角。舵机 Power(一般为红色)接 Arduino 的电源 5V 引脚，Ground(一般为棕色)接 Arduino 的 GND 引脚，Control(一般为橙色)接 Arduino 板的数字引脚 9。

在 Arduino 中使用 Servo.h 库函数驱动舵机，使用 attach()函数绑定舵机控制线，使用

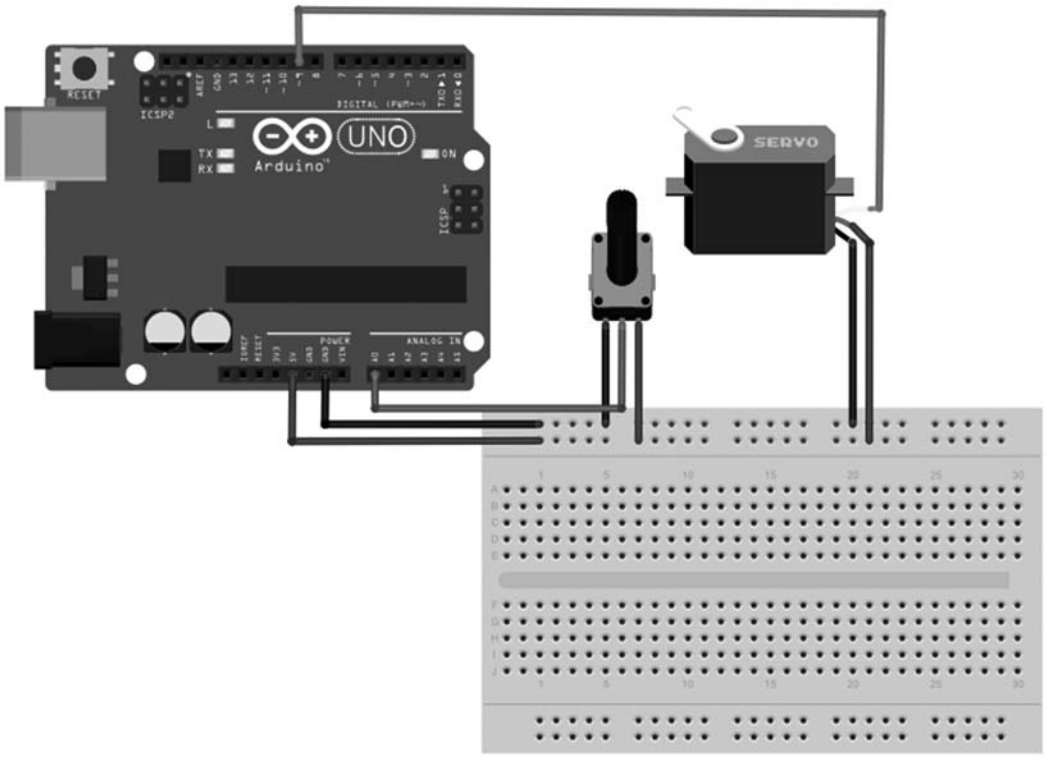


图 5-43　舵机控制电路

write()函数控制舵机的旋转角度，传递参数为 0～180。

```
#include <Servo.h>

Servo myservo;              //定义 Servo 对象来控制
int val;
int potpin = 0;

void setup() {
  myservo.attach(9);        //控制线连接数字 9
}

void loop() {
  val = analogRead(potpin);
  val = map(val,0,1023,0,179);
  myservo.write(val);
  delay(15);
}
```

5.6.3　步进电机

步进电机是将脉冲信号转换成机械运动的一种特殊电机。与伺服电机不同，其不需要额外的反馈即可完成对电机位置与速度的精准控制。它通过脉冲信号在步进电机内部产生

了一个可以旋转的磁场，如图 5-44 所示，当旋转磁场依次切换时，转子(Rotor)就会随之转动相应的角度。但当磁场旋转过快或者转子上所带负载的转动惯量太大时，转子就无法跟上旋转速度，从而造成失步现象。

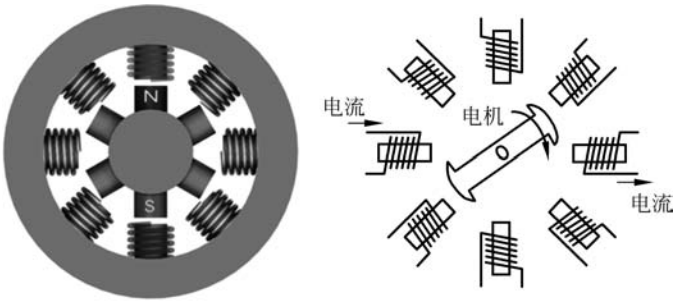


图 5-44　步进电机内部工作原理

步进电机的磁极数量规格和接线规格很多，为简化问题，本书就先只以四相步进电机为例进行讨论。所谓四相，就是指电机内部有 4 对磁极。通常四相电机可以向外引出 6 条接线，包括两个公共端 COM 与 ABCD 接线头，形成六线四相制。也可以将两个 COM 端短接后引出，形成五线四相制，如图 5-45 所示。

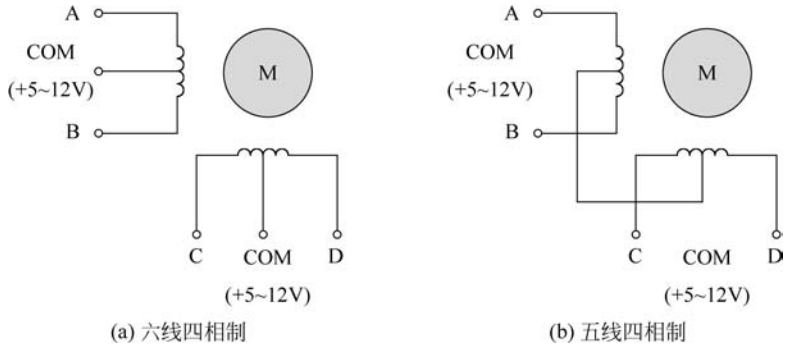


图 5-45　六线四相制与五线四相制

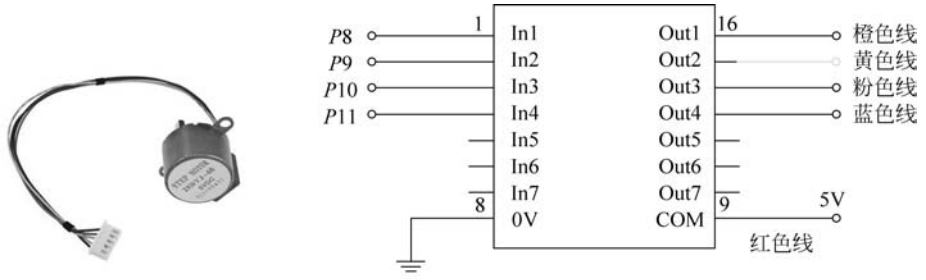
假如某一刻只有一相励磁通电，称为一相励磁方式。励磁通电顺序为 A→B→C→D→A 时，就会在步进电机中产生逆时针旋转的磁场，转子也会逆时针旋转，此方式励磁旋转一周需要 4 步。反之顺序为 D→C→B→A→D，电机则会顺时针旋转。在这种方式下，电机在每个瞬间只有一个线圈导通，消耗电力小但在切换瞬间没有任何的电磁作用转子上，容易造成振动，也容易因为惯性而失步。

假如某一时刻有两相励磁通电，称为二相励磁方式。当励磁通电顺序为 DA→AB→BC→CD→DA 时，会在步进电机内部产生逆时针旋转磁场，此方式励磁旋转一周也需要 4 步。反之则会顺时针旋转。这种方式输出的转矩较大且振动较少，切换过程中至少有一个线圈通电作用于转子，使得输出的转矩较大，振动较小，也比一相励磁较为平稳，不易失步。

综合上述两种驱动信号，提出一相励磁和二相励磁交替进行的方式，即逆时针旋转时励磁通电顺序为 A→AB→B→BC→C→CD→D→DA→A，每传送一个励磁信号，步进电机前进半个步距角，此方式励磁旋转一周需要 8 步。此方式电机旋转角度的分辨率高，运转也更

加平滑。

如图 5-46(a)所示为 28BYJ-48 步进电机，减速比为 64∶1，在 5V 供电电压下转速约为 15r/min，适当升高供电电压可提高其转速。若使用 4 步控制信号序列，则每步旋转 11.25°，在减速机构前 32 步电机旋转 1 周，减速机构后旋转一圈需要 32×64＝2048(步)。若使用 8 步控制信号序列，则每步旋转 5.625 步，64 步电机旋转一周。



(a) 实物图　　(b) 驱动电路

图 5-46　步进电机驱动电路

可以使用 ULN2003 步进电机驱动板来控制 28BYJ-48 步进电机。驱动板的电机供电连接到 Arduino 的 GND 和 5V 取电，使用引脚 8、9、10、11 接 ULN2003A 的 In1、In2、In3、In4，ULN2003A 的输出接步进电机。电路如图 5-46(b)所示。

在 Arduino 中使用 Steeper.h 库函数驱动步进电机，使用 setSpeed()函数设定步进电机旋转速度，即每分钟多少步。step()函数为执行电机驱动多少步，传入正整数则正向旋转，传入负整数则反向旋转。

```
#include <Stepper.h>

//减速前旋转一周需要的步数
const int stepsPerRevolution = 64;

Stepper myStepper(stepsPerRevolution, 8,9,10,11);

int stepCount = 0;

void setup()
{ }

void loop()
{
  int sensorReading = analogRead(A0);
  int motorSpeed = map(sensorReading, 0, 1023, 0, 255);
  if (motorSpeed > 0)
  {
    myStepper.setSpeed(motorSpeed);
    myStepper.step(2048);
    delay(10);
  }
}
```

下载程序后会发现电机将沿着顺时针方向旋转，电位器的模拟量越高，步进电机的转速就越快。

5.7 Arduino 实战——循迹小车

本章前面的内容均只针对 Arduino 中某一模块进行了介绍，无法构成一个完整的系统。因此本节带领读者完成一个较为系统的实验——循迹小车：小车在运动过程中要不断地调整运行状态使车体循着黑色的导引线平稳前进，其中循迹线为黑色、背景为白色。具体轨迹如图 5-47 所示。将小车放置起始位置后，小车开始随着给定线路自动循迹，最终停止在终止线处。

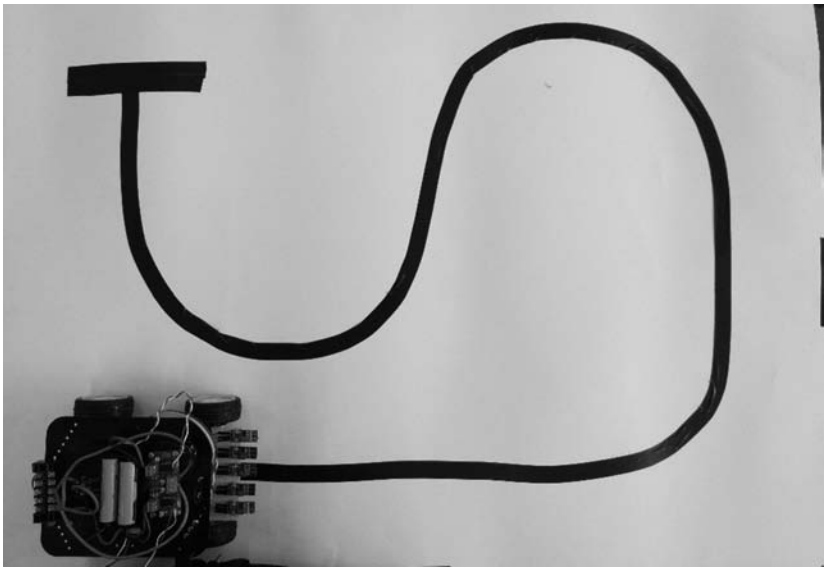


图 5-47 小车运行轨迹

5.7.1 总体方案设计

如图 5-48 所示，循迹小车主要由以下几个部分组成：Arduino、电机驱动模块、黑线检测传感器。通过黑线检测传感器来感知轨迹线与当前车身的偏差，从而及时调整小车的前行方向，最终实现小车按预先给定路线实现自动循迹。

图 5-48 循迹小车组成框图

5.7.2 硬件设计

循迹小车使用的控制器为 Arduino。除此之外，小车的硬件设计还包括 3 个部分：小车车体设计、传感器设计以及电机驱动电路的设计。

1. 车体

对于小车车体，本实验采用 DFRobot A4WD 型号的车体，如图 5-49(a)所示，其分解后各部分如图 5-49(b)所示。

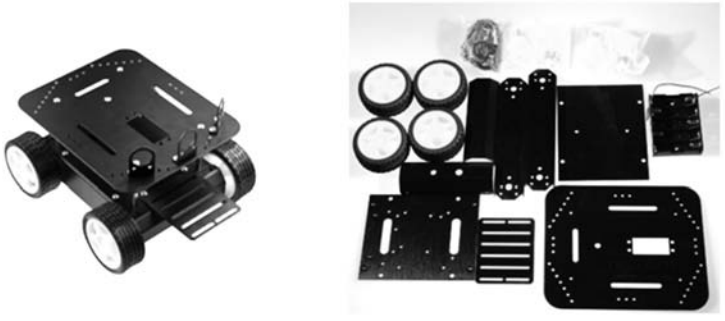


图 5-49　DFRobot A4WD 车体与分解图

2. 传感器设计

对于黑线检测传感器使用之前介绍的红外传感器去探测黑线的位置，其原理在前面已详细介绍过，此处不再赘述。在本实验中共使用 5 个红外传感器来感知小车车身与轨迹线的相对位置，并衡量小车车体与正常轨迹的偏差程度。以左偏为例，如图 5-50 所示，图 5-50(a)～图 5-50(f)所示表明小车左偏程度不断增大。

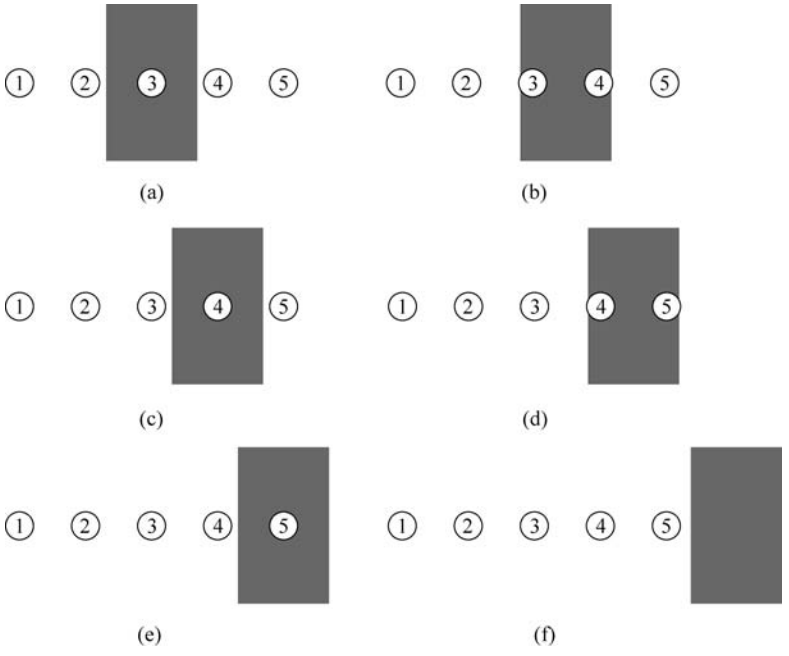


图 5-50　小车左偏的 6 种程度(标明从左到右分别为 1～5)

由于这 5 个传感器输出均为数字信号，因此直接接入 Arduino 对应的 5 个输入引脚即可，如表 5-4 所示。

表 5-4 传感器的接线

红外检测传感器 ID		Arduino 引脚
1	→	A0
2	→	A1
3	→	A2
4	→	A3
5	→	A4

3. 电机驱动电路设计

当小车偏离正常轨迹，则需要及时调整小车前进方向。在 DFRobot A4WD 车体中包含 4 个电机，本实验将左右两个电机各为一组，每侧的两个电机转速相同。当左右两侧电机转速不同或旋转方向不同，即可实现小车的转弯。例如，当左侧电机转速大于右侧或左侧电机正转、右侧电机反转时，小车会右转。这种调整小车前进方向的方式称为差速转向。

采用如图 5-37 所示的电路图，虽然能控制电机的转速，但无法控制电机的转向。为了同时控制左右两侧的电机的转速与转向，本书采用 MC33931 芯片对电机进行控制，控制左侧电机的原理图如图 5-51 所示。

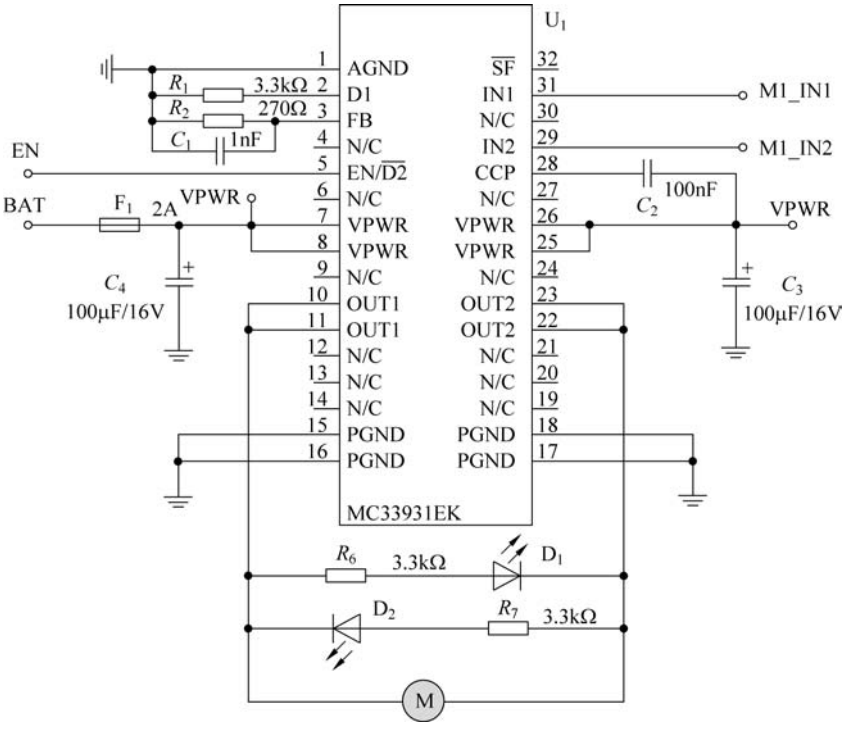


图 5-51 MC33931 电机驱动原理图

其芯片内部事实上为一个 H 桥(也称全桥)结构，如图 5-52 所示。如图 5-52(a)所示，左上角与右下角的 MOS 管同时导通，OUT1 大于 OUT2，此时电机正转。如图 5-52(b)中所示，右上角与左下角的 MOS 管同时导通，OUT1 小于 OUT2，此时电机反转。

可通过 MC33931 的 IN1(31 脚)、IN2(29 脚)、$D1$(2 脚)、EN/$\overline{D2}$(5 脚)对 OUT1 与

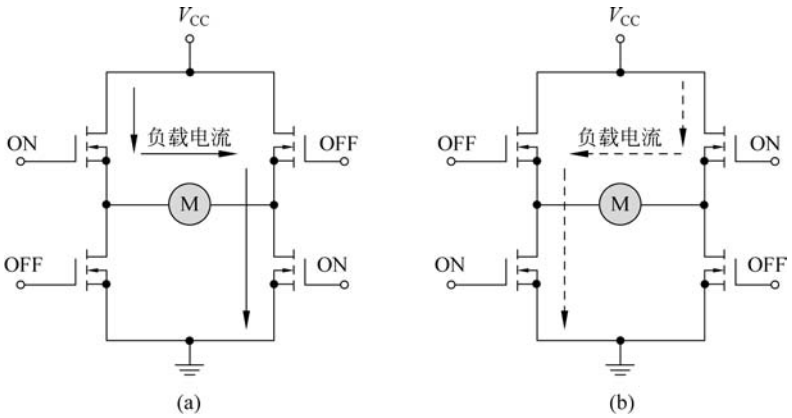


图 5-52　H 全球正反转

OUT2 引脚的电平进行控制，具体如表 5-5 所示。另外 $\overline{SF}$(32 脚)与 FB(3 脚)分别为状态输出与反馈引脚，本书并不使用。

表 5-5　MC33931 芯片输入与输出状态表。H 为高电平、L 为低电平、X 为高或低电平、Z 为高阻态

输　入				输　出		设备状态
EN/$\overline{D2}$	D1	IN1	IN2	OUT1	OUT2	
H	L	H	L	H	L	正转
H	L	L	H	L	H	反转
H	H	X	X	Z	Z	禁止输入模式
L	X	X	X	Z	Z	睡眠模式

由表 5-5 可以看出，在 EN/$\overline{D2}$ 为高电平、$D1$ 与低电平的情况下 MC33931 的 OUT1 与 OUT2 才能驱动电机。显然，IN2 引脚为低电平时，IN1 输入 PWM 波时电机正转，且转速会随着占空比的变化而变化；当 IN1 引脚为低电平，IN2 引脚输入 PWM 波时电机反转，且转速也会随着占空比的变化而变化。电机驱动电路设计图及实物图如图 5-53 所示。

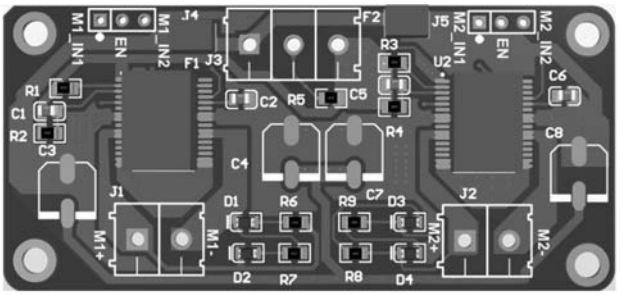
本实验中，左侧驱动电路的 IN1 与 IN2 引脚连接 Arduino 的 PD5 与 PD6，右侧驱动电路的 IN1 与 IN2 引脚分别接入 Arduino 的 PD10 与 PD11。这 4 个引脚均能输出 PWM 波形，通过这 4 个引脚可实现小车任意角度的旋转。EN 引脚连接 Arduino 的 PD8，用于使能与失能控制。

5.7.3　软件设计

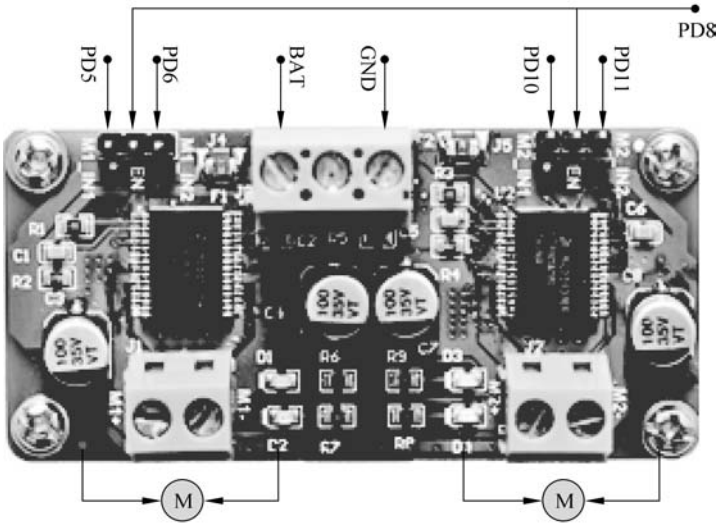
对于软件部分，可分为 3 部分：检测、分析与控制。Arduino 获取检测数据后进行分析，并控制小车动作。

1. 检测

由于本实验采用了 5 个红外传感器，根据这 5 个传感器的值就可判定小车的偏移轨迹是向左还是向右。图 5-50 显示出了小车左偏的 5 种情况以及无偏差的 1 种情况，再加上对应右偏的 5 种情况，可知本系统传感器功能输出 11 种情况，本实验采用－9、－7、－5、－3、－1、0、1、3、5、7、9 这 11 个数字表示偏离轨道的情况，数字大于 0 代表小车左偏，且数字越



(a) 设计图



(b) 实物图

图 5-53　电机驱动原理及实物图

大偏离程度越大；相反地，数字小于 0 代表小车右偏，且数字越大偏离程度也越大。对应程序如下所示。

```
const int IR_PIN[] = {A0, A1, A2, A3, A4};        //红外传感器对应引脚定义
int irs = 0;                                      //保存所有红外传感器的值
bool is_running = true;                           //正在运动标志位
int last_input = 0;                               //传感器的上一次检测结果

void read_ir_values()
{
    irs = 0;                                      //初始化传感器的值
    for (int i = 0; i<5;i++){
        //irs 转换为二进制，每一位代表一个传感器的值
        irs |= digitalRead(IR_PIN[i]) << i;
    }
  switch (irs) {
```

```
        case B00000:
            if (error < 0) {
                if (last_input < -1) error = -9;                    //向右偏离出了轨道
            } else {
                if (last_input > 1)error = 9;                       //向左偏离出了轨道
            }
            break;
        case B00001: error = -7; break;                             //右偏
        case B00011: error = -5; break;                             //右偏
        case B00010: error = -3; break;                             //右偏
        case B00110: error = -1; break;                             //右偏
        case B00100: error = 0; break;                              //无偏差
        case B01100: error = 1; break;                              //左偏
        case B01000: error = 3; break;                              //左偏
        case B11000: error = 5; break;                              //左偏
        case B10000: error = 7; break;                              //左偏
        case B11111:
        is_running = true;                                          //停止
        break;
      }
        last_input = error;
    }
```

2. 分析

当 Arduino 得到小车偏离的数据后，则可根据不同偏离的程度，设定不同的策略，让小车始跟随轨迹运动，形如：

```
if(error == -11)        //右偏程度特别大
    //向左大角度转弯
else if (error == 1)    //轻微左偏
    //向右小角度偏移
//…
```

以上为循迹小车最基本的控制思想，虽然保证小车不偏离出跑道，但如果读者做实验就会发现，小车在行走过程中会不断来回摆动，特别是在过弯情况下表现特别不平稳。这也能说明此时小车缺失根据当前偏离程度动态调整的能力，导致最终控制效果不好。因此，可借助控制领域常用的 PID 算法进行优化。

PID 算法同时将过去、现在以及将来可能发生的误差同时进行分析，得出更适合当前状态下的运动参数。本实验使用的全量式 PID 算法，具体代码如下所示。

```
/* PID 参数以及初始值设定 */
float Kp = 10, Ki = 0.002, Kd = 1;
float error = 0, P = 0, I = 0, D = 0, PID_value = 0;
float previous_error = 0;

/* PID 计算 */
void calculate_pid()
{
```

```
//求得式(5-4)中各项参数
    P = error;                          //P 为式(5-4)中比例部分的 e_k
    I = I + error;                      //所有 e_k 的求和
    D = error - previous_error;         //与上一个误差的差值,即式(5-4)中的 e_k - e_k-1
    //计算 PID 输出值
    PID_value = (Kp * P) + (Ki * I) + (Kd * D);
    PID_value = constrain(PID_value, -100, 100);     //限制输出范围在[-100,100]之间
    //保存当前误差作为下一次使用
    previous_error = error;
}
```

3. 控制

在分析阶段,PID 算法得出了输出值(PID_value),由于误差大于 0 时,表明小车左偏于轨迹,左侧电机应该加速、右侧轮子应该减速。因此对于左侧电机实际速度应等于期望速度加上输出值,右侧电机则正好相反。因此很容易得到如下代码。

```
const int M_L_IN1 = 5;                          //电机 A1
const int M_L_IN2 = 6;                          //电机 A2
const int M_R_IN1 = 10;                         //电机 B1
const int M_R_IN2 = 11;                         //电机 B2
const int EN_PIN = 8;                           //使能引脚

static int initial_motor_speed = 80;            //期望速度
int left_motor_speed = 0;
int right_motor_speed = 0;

/* 左侧的电机控制 */
void left_forward_run(int m_speed)              //左侧电机正转,可根据实际接线修改
{
  analogWrite(M_L_IN1, m_speed);
  analogWrite(M_L_IN2, 0);
}

void left_reversal_run(int m_speed)             //左侧电机反转,可根据实际接线修改
{
   analogWrite(M_L_IN1, 0);
   analogWrite(M_L_IN2, m_speed);
}

/* 右侧的电机控制 */
void right_forward_run(int m_speed)             //右侧电机正转,可根据实际接线修改
{
   analogWrite(M_R_IN1, m_speed);
   analogWrite(M_R_IN2, 0);
}

void right_reversal_run(int m_speed)            //左侧电机反转,可根据实际接线修改
{
```

```
    analogWrite(M_R_IN1, 0);
    analogWrite(M_R_IN2, m_speed);
}

/* 电机的使能与失能 */
void enable_run()
{
    digitalWrite(EN_PIN, HIGH);
}

void disenable_run()
{
    digitalWrite(EN_PIN, LOW);
    analogWrite(M_L_IN1, 0);
    analogWrite(M_L_IN2, 0);
    analogWrite(M_R_IN1, 0);
    analogWrite(M_R_IN2, 0);
    delay(1);
}

void motor_control()
{
    //计算每个电机的速度
    left_motor_speed = initial_motor_speed - PID_value;
    right_motor_speed = initial_motor_speed + PID_value;
    constrain(left_motor_speed, -155,155);          //速度限定在(-155,155)
    constrain(right_motor_speed, -155, 155);
    if (left_motor_speed > 0) {                     /** 左侧 **/
        left_forward_run(left_motor_speed);         //正转
} else {
        left_reversal_run(abs(left_motor_speed));   //反转
    }
    if (right_motor_speed > 0) {/** 右侧 **/
        right_forward_run(right_motor_speed);       //正转
    } else {
        right_reversal_run(abs(right_motor_speed)); //反转
    }
}
```

4. 主函数编写

在主函数中相对比较简单，依次调用之前的检测、分析与控制代码即可，如下所示。

```
void setup()                                        //初始化
{
    pinMode(EN_PIN, OUTPUT);
    disenable_run();
    pinMode(M_L_IN1, OUTPUT);
    pinMode(M_L_IN2, OUTPUT);
    pinMode(M_R_IN1, OUTPUT);
```

```
    pinMode(M_R_IN2, OUTPUT);
    for (int i = 0; i < 5; i++) {
        pinMode(IR_PIN[i], INPUT);
    }
    right_forward_run(initial_motor_speed);          //正转
    left_forward_run(initial_motor_speed);           //正转
    delay(3000);                                     //通电 3s 后自动启动
    enable_run();                                    //启动
}

void loop()
{
    read_ir_values();                                //读取传感器的值
    if (is_running){
        calculate_pid();                             //计算 PID 值
        motor_control();                             //根据 PID 控制电机
}else{
        analogWrite(M_L_IN1, 0);
        analogWrite(M_L_IN2, 0);
        analogWrite(M_R_IN1, 0);
        analogWrite(M_R_IN2, 0);
        while(1);                                    //停止
    }
}
```