

第3章

Arduino语言



3.1 Arduino 使用的语言

Arduino 使用 C/C++ 编写程序。虽然 C++ 兼容 C 语言,但却 是两种语言,C 语言是一种面向过程的编程语言,C++ 是一种面 向对象的编程语言。早期的 Arduino 核心库使用 C 语言编写,后 来引进了面向对象的思想,目前最新的 Arduino 核心库采用 C 与 C++ 混合编写而成。

通常所说的 Arduino 语言,是指 Arduino 核心库文件提供的 各种应用程序编程接口(Application Programming Interface,API) 的集合。这些 API 是对更底层的单片机支持库进行二次封装所形 成的。

3.2 Arduino 程序结构

如图 3-1 所示,Arduino 程序的基本结构由 `setup()` 和 `loop()` 两个函数组成。

```
void setup() {  
    // put your setup code here, to run once:  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

图 3-1 Arduino 程序结构

1. setup()

Arduino 控制器通电或复位后,即会开始执行 setup()函数中的程序,该程序只会执行一次。通常是在 setup()函数中完成 Arduino 的初始化设置。

2. loop()

setup()函数中的程序执行完毕后,Arduino 会接着执行 loop()函数中的程序。而 loop()函数是一个死循环,其中的程序会不断地重复运行。通常是在 loop()函数中完成程序的主要功能,如采集数据和驱动模块等。

3.3 数字 I/O 口的操作函数

数字操作即为高低电平操作(0/1)。

```
pinMode(pin,mode)
```

设置引脚模式,mode 有 OUTPUT(输出),INPUT(输入)。

```
digitalWrite(pin,value)
```

设置引脚的输出电平(高低),value 为高低电平(HIGH/LOW 或 1/0)。其中,低电平的电压为 0V,高电平的电压为 5V。

```
digitalRead(pin)
```

读取输入引脚的电平情况。当 Arduino 以 5V 供电时,会将范围为 $-0.5\sim 1.5\text{V}$ 的输入电压作为低电平识别,而将范围在 $3\sim 5.5\text{V}$ 的输入电压作为高电平识别。

在 Arduino 核心库中: OUTPUT 被定义为 1; INPUT 被定义为 0; HIGH 被定义为 1; LOW 被定义为 0。

3.4 模拟 I/O 口的操作函数

在 Arduino 控制器中,编号前带有“A”的引脚是模拟输入引脚。

模拟 I/O 口可以将 $0\sim 5\text{V}$ 的电压转换为 $0\sim 1023$ 的整数形式表示。

```
analogRead(pin)
```

读取指定引脚的模拟值。pin 必须是模拟输入引脚。

```
analogWrite(pin,value)
```

其中,pin 是要输出 PWM 波(脉冲宽度调节)的引脚,value 是 PWM 的脉冲宽度,范围为 $0\sim 255$ 。

3.5 延时函数

使用延时函数 `delay()` 可以暂停程序,并通过参数来设定延时时间,用法如下:

```
delay();
```

此函数是毫秒级延时,参数的数据类型为 `unsigned long`。

3.6 常用的数据类型

1. char 字符型

字符型,即 `char` 类型,占用 1 字节的内存空间,主要用于存储字符变量。在存储字符时,字符需要用单引号引用,例如:

```
Char col = 'c'
```

字符都是以数字形式存储在 `char` 类型变量中的,数字与字符的对应关系可参照 ASCII 码表。

2. int 整型

整型即整数类型,取值范围为 $-32768 \sim 32767 (-2^{15} \sim 2^{15} - 1)$ 。

3. float 单精度浮点型

`float` 即浮点型数据类型,浮点数其实就是通常所说的实数。

浮点型数据运算较慢且有一定误差,因此,通常会把浮点型转

换为整型来处理。

3.7 常用的控制语句

1. if 语句

if 语句是最常用的选择结构实现方式,当给定的表达式为真时,就会运行其后的语句。

(1) 简单分支结构。

```
if (表达式)
{
    语句;
}
```

(2) 双分支结构。

if...else,当给定的表达式为假时,则运行 else 后的语句。

```
if(表达式)
{
    语句 1;
}
else
{
    语句 2;
}
```

(3) 多分支结构。

将 if 语句嵌套使用,即形成多分支结构,以判断多种不同的情况。

```
if(表达式 1)
```

```
{  
    语句 1;  
}  
else if(表达式 2)  
{  
    语句 2;  
}  
else if(表达式 3)  
{  
    语句 3;  
}  
.....
```

2. switch...case 语句

当处理比较复杂的问题时,可能会存在有很多选择分支的情况,如果还使用 if...else 的结构编写程序,会使程序显得冗长,且可读性差。

此时可以使用 switch...case 语句,其一般形式为:

```
switch(表达式)  
{  
    case 常量表达式 1;  
        语句 1  
        break;  
    case 常量表达式 2;  
        语句 2  
        break;  
    case 常量表达式 3;  
        语句 3  
        break;  
    .....  
    default :
```

```
    语句 n  
    break;  
}
```

需要注意的是,switch 后的表达式的结果只能是整型或字符型,如果使用其他类型,则必须使用 if 语句。

switch 结构会将 switch 语句后的表达式与 case 后的常量表达式进行比较,如果相符就运行常量表达式所对应的语句;如果不符则会运行 default 后的语句;如果不存在 default 部分,程序将直接退出 switch 结构。

在进入 case 判断并执行完相应程序后,一般要使用 break 语句退出 switch 结构。如果没有使用 break 语句,则程序会一直执行到有 break 的位置,才会退出或运行完 switch 结构再退出。

3. 循环语句

(1) while 循环。

while 循环是一种“当”型循环。当满足一定条件后,才会执行循环体中的语句,其一般形式为:

```
while(表达式)  
{  
    语句;  
}  
do...while 循环
```

do...while 循环与 while 循环不同,是一种“直到”循环,它会一直循环到给定条件不成立为止。do...while 会先执行一次 do 语句后的循环体,再判断是否进行下一次循环,即

```
do
{
    语句;
}
while(表达式);
```

(2) for 循环。

for 循环比 while 循环更灵活,而且应用广泛。它不仅适用于循环次数确定的情况,也适用于循环次数不确定的情况。while 和 do...while 都可以替换为 for 循环。其一般形式为:

```
for(表达式 1; 表达式 2; 表达式 3; )
{
    语句;
}
```

在一般情况下,表达式 1 为 for 循环初始化语句,表达式 2 为判断语句,表达式 3 为增量语句。

4. 循环控制语句

(1) break。

break 语句只能用于 switch 多分支选择结构和循环结构中,使用它可以终止当前的选择结构或者循环结构,使程序转到后续的语句运行。break 一般会搭配 if 语句使用,其一般形式为:

```
if(表达式)
{
    break;
}
```

(2) continue。

continue 语句用于跳过本次循环中剩下的语句,并且判断是否开始下一次循环。同样,continue 一般搭配 if 语句使用,其一般形式为:

```
if(表达式)
{
    continue;
}
```

3.8 相关语法

1. 注释

“/*”与“*/”之间的内容以及“//”之后的内容均为程序注释,使用它们可以更好地管理代码。注释不会被编译到程序中,因此不影响程序的运行。

为程序添加注释的方法有以下两种。

(1) 单行注释,语句为:

```
//注释内容
```

(2) 多行注释,语句为:

```
/*
注释内容 1
注释内容 2
.....
*/
```

2. define

define 即宏定义,即使用一个特定的标识符来代表一个字符串。宏定义的一般形式为:

```
#define 标识符字符串
```

3. include

“include”意为“包含”。若程序中使用 #include 语句包含了一个文件,例如 #include <EEPROM.h>,那么在预处理时,系统会将该语句替换成 EEPROM.h 文件中的实际内容,然后再对替换后的代码进行编译。文件包含命令的一般形式为:

```
#include <文件名>  
或 #include "文件名"
```

两种形式的效果是一样的,只是当使用<文件名>形式时,系统会优先在 Arduino 库文件中寻找目标文件,若没有找到,再到当前 Arduino 项目的项目文件夹中查找。而使用"文件名"形式时,系统会优先在 Arduino 项目的项目文件夹中查找目标文件,若没有找到,再查找 Arduino 库文件。