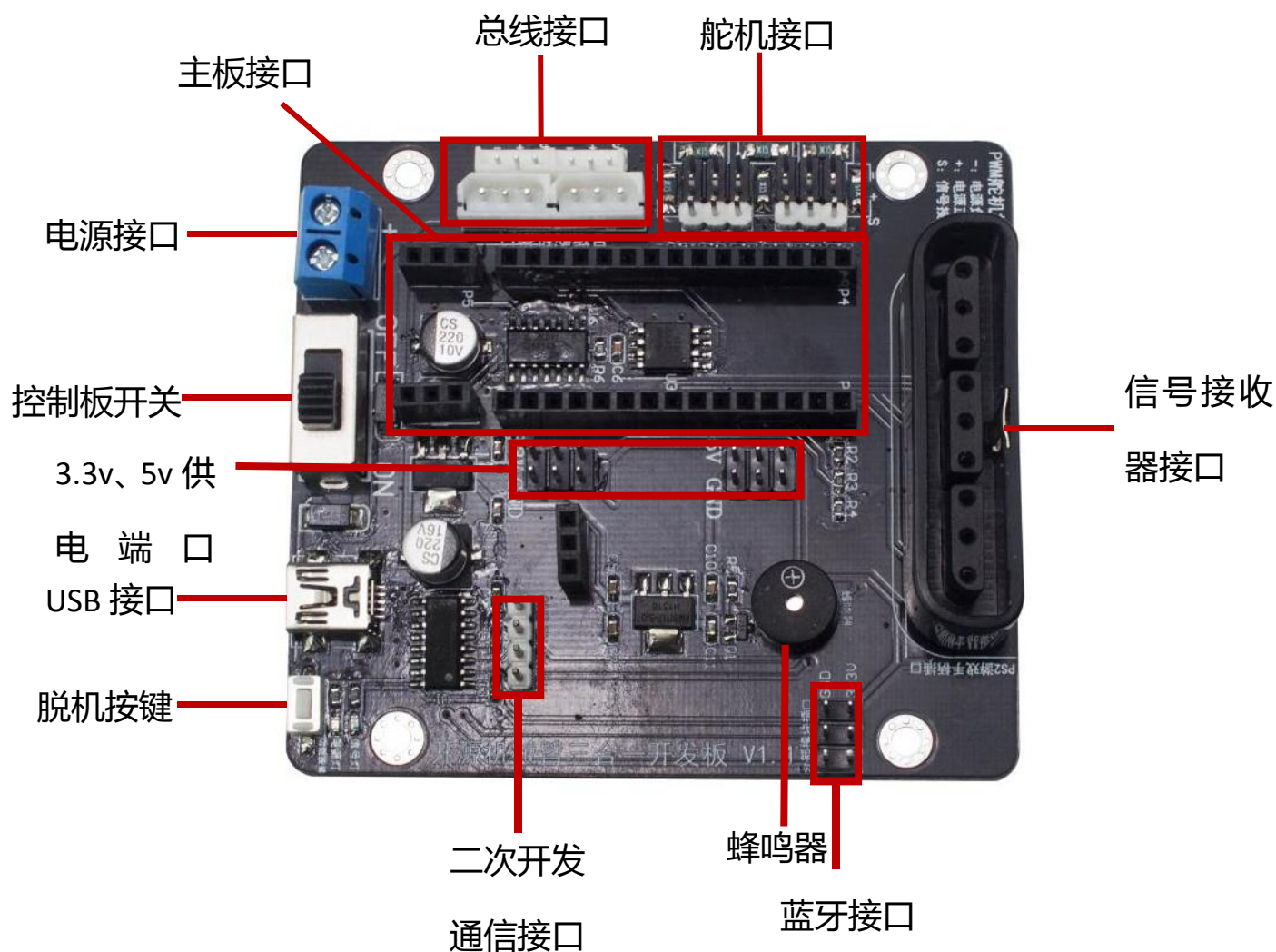




机械臂介绍

➤ 控制板



控制板包括两个部分，一个是基础板，如上图示，还有一部分就是接在主板接口的 Arduino 板、51 板或者 STM32 板。控制板的正常工作电压在 6.4-8.4v，低于 6.4v 时蜂鸣器会发出低压报警信号，提醒客户及时充电，不过我们供电采用的是 AD/DC 电源适配器，只要插上就不用担心电压的问题。控制板上的 USB 接口是用来下载程序，动作组或者调试用的，只需接到电脑上，打开相应的上位

机即可。脱机按键，当按下时机械臂会运行 100 号动作组（动作号可在程序中修改）。总线接口，接串口舵机，舵机接口接 PWM 舵机。信号接收器端口接信号接收器，客户可通过手柄控制机械臂。中间的 3.3v，5v 端口是控制板给外接设备可提供的电压值。二次开发通信接口是用来与其他单片机通信的，蓝牙接口可接蓝牙通信设备，然后通过手机连接控制机械臂。

➤ 舵机

整套机械臂一共使用了 6 个高精度数字舵机

我们从上往下看。爪子部分使用的是具有防堵转功能的 LDX-335MG 数字舵机，当发生堵转时，舵机会自动计时，当发现堵转超过 4 分钟时，舵机会自动停止工作。接下来两个舵机是

LFD-06 防堵转低功耗的数字舵机，当发生堵转时，舵机内部会自动进行保护。在云台上的两个舵机是高精度的双轴数字舵机，它们采用插拔的连接方式，布线、更换都十分方便。最后底座上的舵机采用大扭力的 1501 舵机，它有 15KG 的扭力，能旋转 180°。

➤ 支架结构

主体支架呈蓝色，全部采用硬铝合金，加上表面喷砂氧化处理，质感好，硬度高，

注意事项

- ① 开启机械臂时、运行机械臂时请和机械臂保持一定的距离，保证安全。
- ② 机械臂上电后，请不要掰动任意的关节，以免损坏舵机。
- ③ 不用机械臂时请关闭电源，拔掉电源线。
- ④ 51 控制板下载拔掉跳线帽时，蜂鸣器会滋滋响是正常现象
- ⑤ 下载程序时如果不成功，请重启电源多次尝试。

Arduino 版

1 开发环境搭建及烧写程序

1.1 基础知识

Arduino IDE 基于 Processing IDE 开发，可以在 Windows Mac OSX 和 Linux 三大主流操作系统上运行。Arduino 语言是基于 Wiring 语言开发，是对 AVR-GCC 库的二次封装，并不需要太多的单片机基础和编程基础，只要简单地学习后就可以快速的进行开发。

1.2 环境搭建

1.7.1 下载配置 Arduino 开发环境

在开始使用 Arduino 之前，需要在电脑上安装 Arduino 的集成开发环境(此后简称 IDE)。安装包在文件夹 xx 下，只需要打开照着提示安装即可。

1.7.2 认识 Arduino IDE

打开 Arduino IDE 时，可以看到一个简单明了的窗口。如下图所示

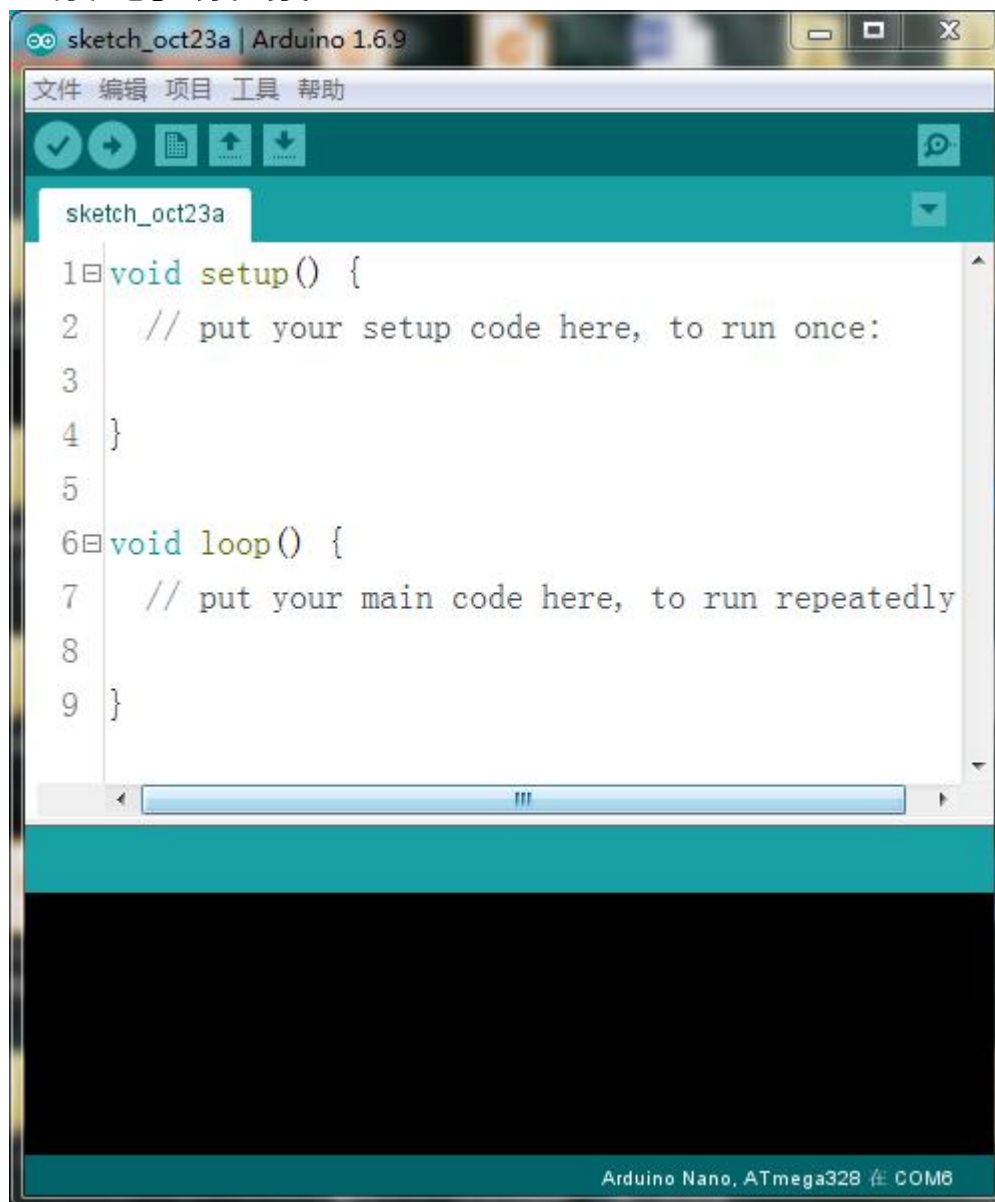


图 1.1 Arduino IDE 界面

选择文件→首选项，在弹出的窗口中设置 IDE 项目保存位置，语言，字体大小，行号等。

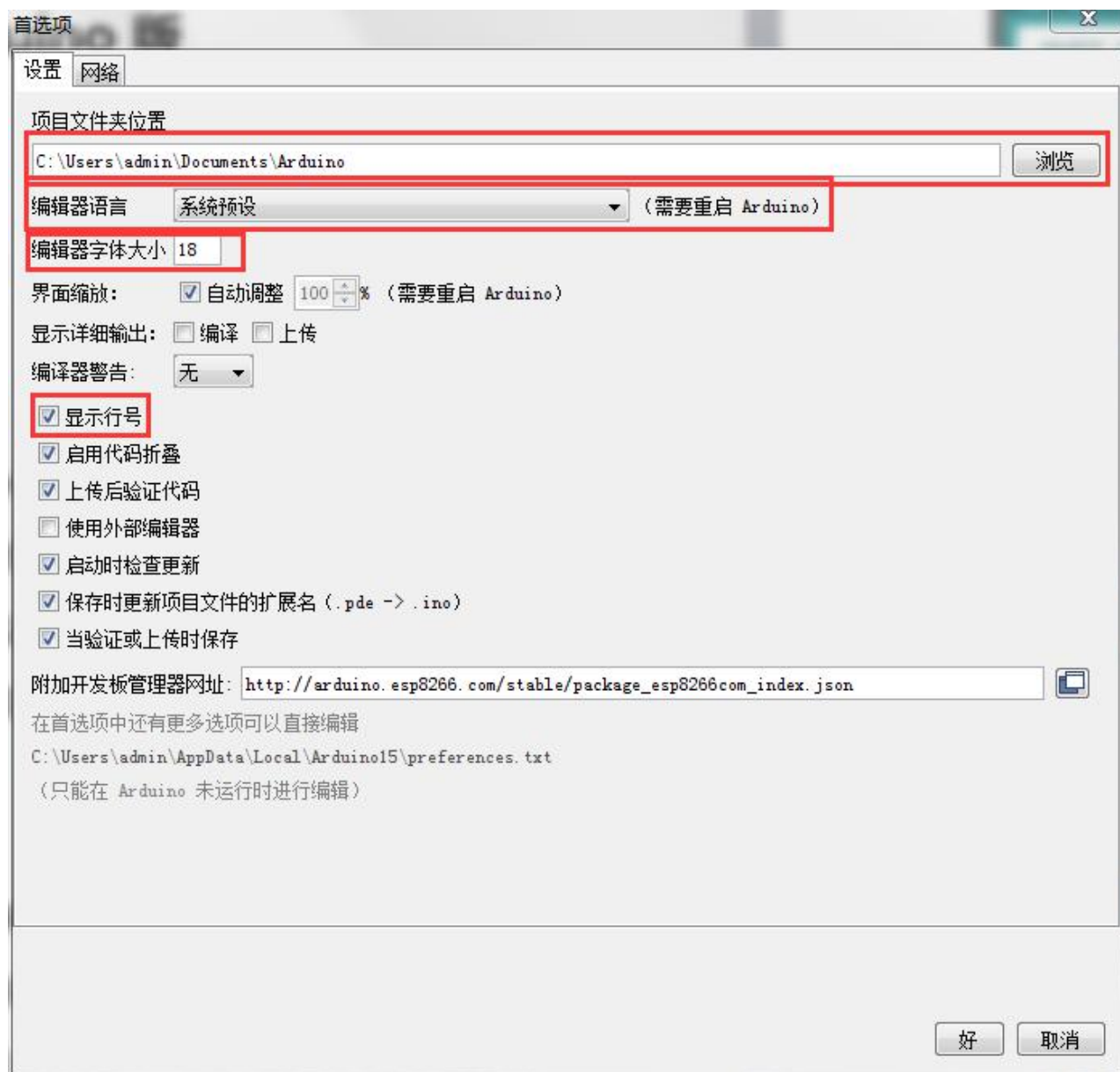






图 1.2 Arduino IDE 设置

Arduino IDE 窗口分为下图所示的几个区域



图 1.3 Arduino IDE 界面功能解析

在工具栏上，Arduino IDE 提供了常用的功能的快捷键：

-  校验，验证一个程序是否编写无误，若无误则编译该项目。
-  下载，下载程序到 Arduino 控制器上。
-  新建，新建一个项目。
-  打开，打开一个项目。



保存，保存一个项目。



串口监视器，IDE 自带的一个简单的串口监视程序，用它可以查看串口发送或接收到的数据。

1.7.3 安装 Arduino 驱动程序

在 Windows 中安装驱动的方法如下：

- ① 用 USB 线连接 Arduino 后，右击选择“计算机”→“属性”→“设备管理器”打开设备管理器界面，在端口这个选项打开，会看到一个“未知设备”。
- ② 双击“未知设备”，并单击“更新驱动程序”按钮。
- ③ 在弹出的对话框中单击“浏览计算机以查找驱动程序软件”。
- ④ 选择驱动所在的地址（即 Arduino 安装目录下的 drivers 文件夹），并单击“下一步”按钮，开始安装驱动。
- ⑤ 安装成功后在设备管理器中可以看到 Arduino 控制器对应的 COM 口。记下该串口号，后面会用到它。

1.3 程序讲解

这里讲一下 Arduino 程序的基本框架

```

1 void setup() {
2     // put your setup code here, to run once:
3
4 }
5
6 void loop() {
7     // put your main code here, to run repeatedly:
8
9 }

```

图 1.4 Arduino 程序基本框架

如上图所示，void setup 和 void loop 是 Arduino IDE 为用户自动生成的程序，同时它们也是一个程序必不可少的两个部分，两个函数里面可以为空但是一定要有。setup 部分是程序开始运行时运行的地方，它会且只运行一次，所以这部分一般都是一些初始化的步骤。然后 loop 里相当于 51 的 main 部分，在运行完 setup 一次后就会一直运行 loop 部分，所以此处是放主程序的地方。

Arduino 还有一个比较方便的地方就是：它有许多库函数可以使用。库函数库函数；顾名思义是把函数放到库里面，是别人把一些常用到的函数编完放到一个文件里，供别人用。别人用的时候把它所在的文件名用 #include<> 加到里面就可以了。

下面讲一下怎么离线以及在线添加库函数

- ① 我们单击“项目”→“加载库”→“管理库”

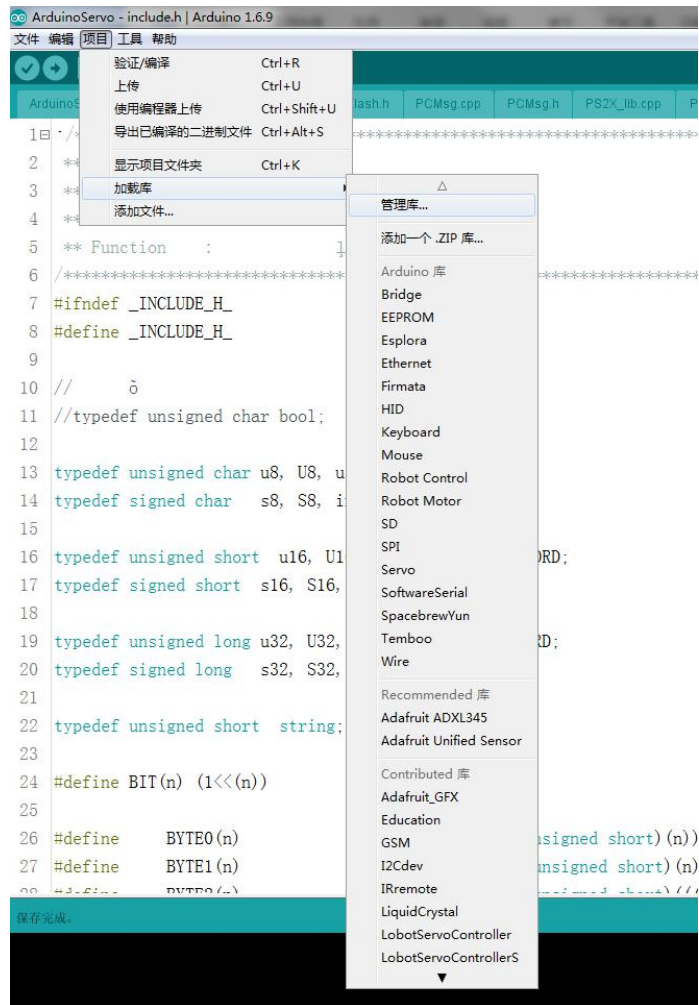


图 1.5 在线添加库

② 在弹出的框中,我们可以在第一行的搜索栏中输入我们需要下载的库函数名,然后在搜索出的结果中找到我们需要的那个,选中点击安装即可。

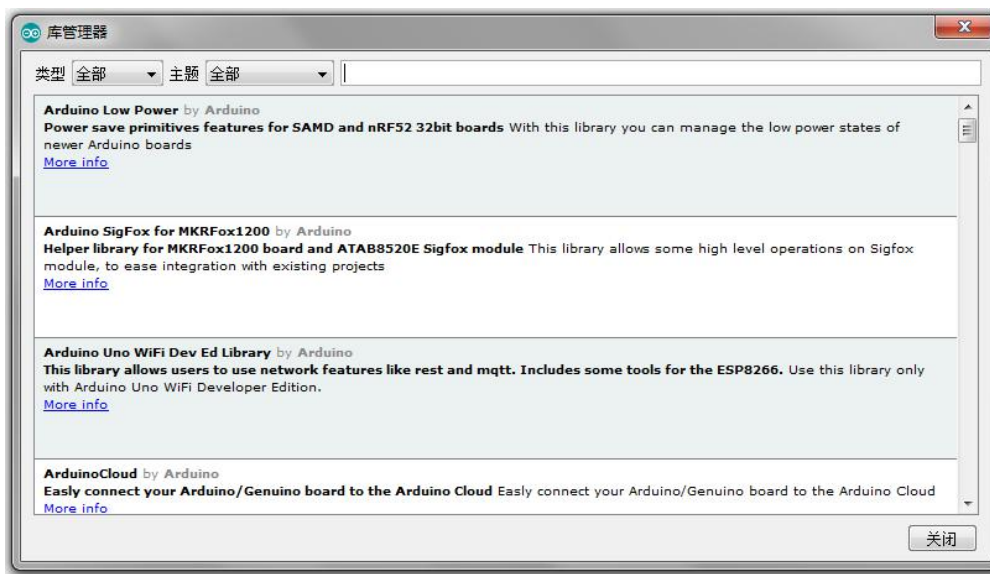


图 1.6 搜索库

③ 离线加载库就是我们事先在网络上下载库的压缩包，然后直接加载进 Arduino 中，同样的我们单击“项目”→“加载库”→“添加一个.zip 库”

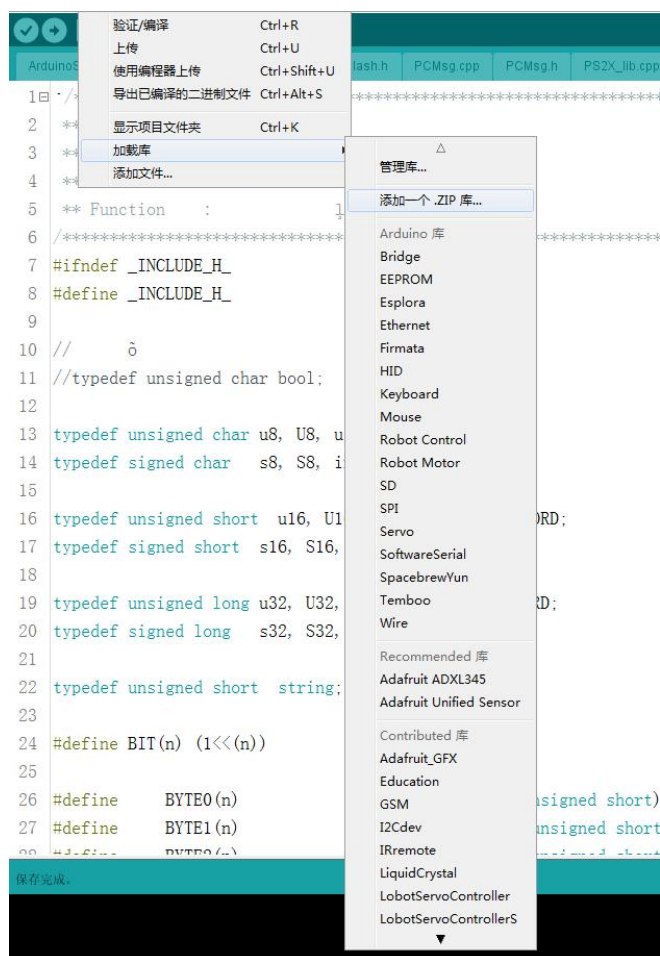


图 1.7 离线添加库

④ 找到自己下载的库文件，加载进去

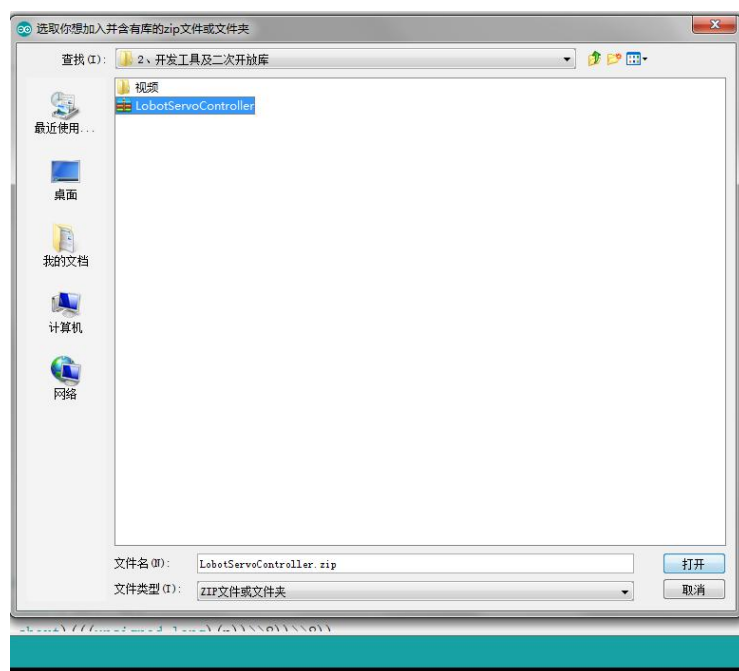


图 1.8 加载本地库

当我们把库加载进 Arduino 时，怎么使用呢？我们还需把库函数加进程序里面，我们在需要用到库的程序中单击“项目”→“加载库”，然后在列表中选择我们需要的那个库，点击它，它就会以`#include<>`的形式显示在程序中，表示已经加载到程序里，当然我们也可以手动输入来加载库，前提是一定要在线或离线加载库到 Arduino 里了。

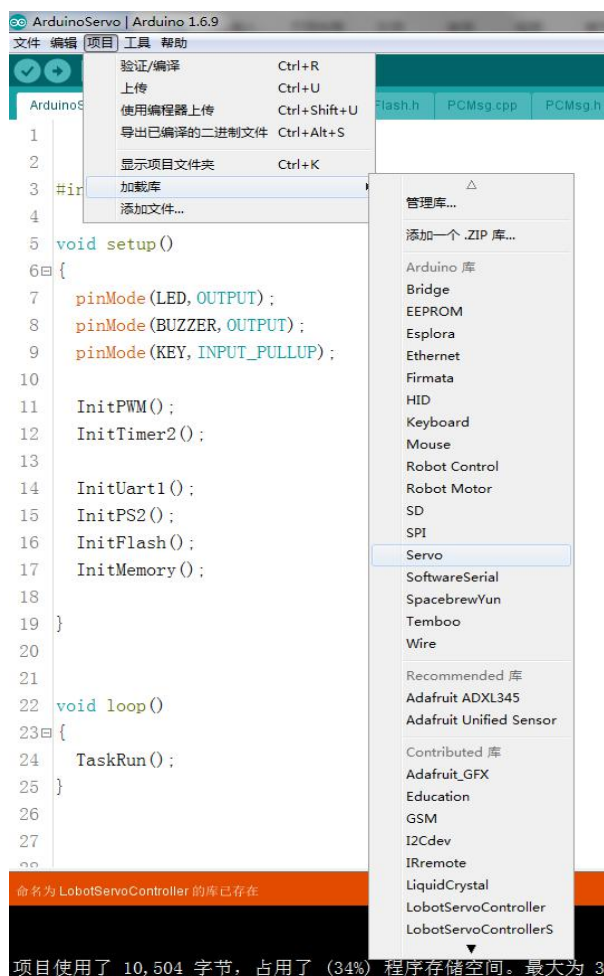


图 1.9 加载库到程序里

1.4 烧录程序

在下载程序前需要在“工具”→“开发板”中选择正在使用的 Arduino 控制器型号，接着在“工具”→“端口”中选择 Arduino 控制器对应的串口，可在设备管理器中查看。设置完后我们点击校验按钮，如果程序无误，则调试区会依次显示“编译程序中”和“编译完毕”。

如有错误会显示相应的错误信息。编译无误后我们点击下载按钮，然后我们看左下角的提示，当显示上传时我们马上按下 Arduino nano 板上的白色复位

按钮，下载完程序会显示上传成功。

2 定时器

2.1 实验器材

控制板

2.2 基础知识

定时器是单片机上用于计算时间的硬件设备。使用定时器可以使单片机以指定的间隔执行指定的操作，或是精确计算事件发生的时间间隔。单片机上定时器一般由基发生器和计数器。

时基即时间的基本单位。时基发生器能产生以时基为周期的信号，计数器则能计数时基发生器产生的信号个数。例如，时基为 1 秒，则时基发生会每秒产生一个信号，计数器的数值则会每秒加 1。当计数器数值等于我们设定的数值时，单片机就会执行我们设定的操作。

2.3 程序讲解

```

18 }
19 void InitTimer2(void)    //100us@12.000MHz
20 {
21     TCCR2A=0;
22     TCCR2B=_BV(CS21)|_BV(CS20); //
23     TIMSK2=_BV(TOIE2);
24     TCNT2=206; //中断时间=(256-206)/500000=100us
25     sei();
26 }
--

```

图 2.1 初始化定时器

在 Arduino 中使用定时器一般都是用封装好的库函数，但是此处我们根据它的数据手册来使用寄存器进行定时器设置，TCCR2A 全部设置为 0 的话就是正常端口操作，TCCR2B 的 CS22 : CS21 : CS20 = 1 : 1: 0，表示 256 分频；TIMSK2 设置成输出比较匹配 A 中断使能，TCNT2 设置初始化时间，最后 Sei 打开总中断。

3 检测按键

2.1 实验器材

控制板

2.2 基础知识

机械式按键在按下或释放时，由于机械弹性作用的影响，通常伴随有一定时间的触点机械抖动，然后其触点才稳定下来，抖动时间一般为 5-10ms，如图 3.1 所示。在触点抖动期间检测按键的通与断状态，可能导致判断出错。

按键的机械抖动可采用硬件电路来消除,也可以采用软件方法进行去抖。

此处我们采用软件去抖的方法。软件去抖的大概思路是这样的：在检测到有键按下时，先执行 10ms 左右的延时程序，然后再重新检测该键是否仍然按下，以确认该键按下不是因抖动引起的。同理，在检测到该键释放时，也采用先延时再判断的方法消除抖动的影响。

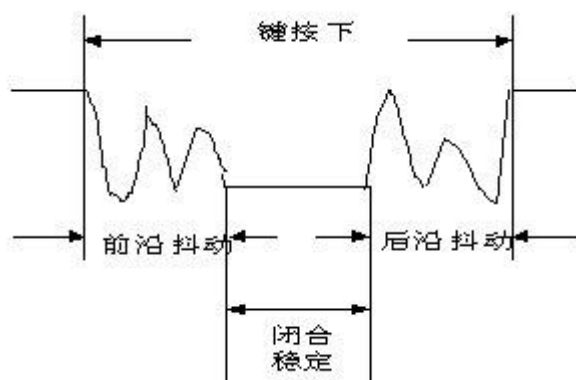


图 3.1 按键触点的机械抖动

2.3 程序讲解

我们看下程序，这个程序在 App.cpp 内，具体如下图所示

```

256
257     if(digitalRead(KEY) == LOW)
258     {
259         delay(60);
260     {
261         if(digitalRead(KEY) == LOW)
262         [
263         //         LED = ~LED;
264             LedFlip();
265             FullActRun(100, 1);
266         }
267     }
268 }
269

```

图 3.2 按键检测程序

首先我们判断按键是否被按下，如果是则延时 60ms 再检测按键是否被按下，如果按键还是处于被按下状态，我们就判定按键被放下，然后执行相应操作：闪烁一次 LED 灯，然后运行 100 号动作组一次。

4 让蜂鸣器响起来

4.1 实验器材

控制板

4.2 基础知识

定时器是单片机上用于计算时间的硬件设备。使用定时器可以使单片机以指定的间隔执行指定的操作，或是精确计算事件发生的时间间隔。 单片机上定时

器一般由基发生器和计数器。

时基即时间的基本单位。时基发生器能产生以时基为周期的信号，计数器则能计数时基发生器产生的信号个数。例如，时基为 1 秒，则时基发生会每秒产生一个信号，计数器的数值则会每秒加 1。当计数器数值等于我们设定的数值时，单片机就会执行我们设定的操作。

综上，让蜂鸣器响起来，就要单片机每隔一段时间就检测一次按键状态。我们要先设定定时器的时基，然后设定计数的值，然后设定到达指定时间后要执行的操作。

4.3 程序讲解

此程序在 App.cpp 内，如下所示

```

64 void Buzzer(void)
65 { //放到100us的定时中断里面
66     static bool fBuzzer = FALSE;
67     static uint32 t1 = 0;
68     static uint32 t2 = 0;
69     if(fBuzzer)
70     {
71         t1++;
72         if(t1 <= 2)
73         {
74             digitalWrite(BUZZER, LOW); //2.5KHz
75         }
76         else if(t1 <= 4)
77         {
78             digitalWrite(BUZZER, HIGH); //2.5KHz
79         }
80         if(t1 == 4)
81         {
82             t1 = 0;
83         }
84     }
85

```

```

86     if(BuzzerState == 0)
87     {
88         fBuzzer = FALSE;
89         t2 = 0;
90     }
91     else if(BuzzerState == 1)
92     {
93         t2++;
94         if(t2 < 5000)
95         {
96             fBuzzer = TRUE;
97         }
98         else if(t2 < 10000)
99         {
100             fBuzzer = FALSE;
101         }
102         else
103         {
104             t2 = 0;
105         }
106     }
107 }
108

```

图 4.1 蜂鸣器函数

我们可以看到刚开始时 fBuzzer = FLASE , BuzzerState = 0 , 所以直接跳过这些程序，这个程序什么时候会运行我们看下面这个程序

```

109 ISR(TIMER2_OVF_vect)
110 {
111     TCNT2=206; //定时器3中断 100us
112
113     static uint16 time = 0;
114     static uint16 timeBattery = 0;
115
116     Buzzer();
117     if(++time >= 10)
118     {
119         time = 0;
120         gSystemTickCount++;
121         // Ps2TimeCount++;
122         if(GetBatteryVoltage() < 6400) //小于6.4V报警
123         {
124             timeBattery++;
125             if(timeBattery > 5000) //持续5秒
126             {
127                 BuzzerState = 1;
128             }
129         }
130         else
131         {
132             timeBattery = 0;
133             BuzzerState = 0;
134         }
135     }
136 }

```

图 4.2 定时器 3 的中断

它是定时器 3 的中断，每 100 微妙会响应一次，它会调用 Buzzer 这个函数，就是我们刚才说的那个函数，但是我们说过了它的两个判断值都不符，直接跳过了。然后我们往下看，延时 $100 \times 10\mu\text{s}$ ，即 1ms 进入 if，获取电池电压

并判断，小于 6400 就持续 5 秒检测，如果还是小于 6400，则将 BuzzerState 置 1，否则，清零重新判断。所以，如果电池电压小于 6.4v，则 BuzzerState 置 1，调用 Buzzer 时进入相应判断，将 fBuzzer 置 true，然后进入第一个 if 置蜂鸣器引脚为高低电平，模拟脉冲，让蜂鸣器响起来。持续一段时间后，关闭蜂鸣器，然后再去判断电池电压，如此循环。

5 ADC 检测电池电压并实现低压报警

5.1 实验器材

控制板、电池

5.2 基础知识

ADC 即 A/D Converter 是模数转换器的简称，在单片机应用系统中，经常需要把输入的模拟电压信号转换为单片机能够识别的数字信号，将连续变化的模拟信号转换为数字信号的技术称为 A/D 转换技术。在实际应用中，可以在输入信号与单片机之间连接 A/D 转换器来完成 A/D 转换，还可以选择使用具有内部 A/D 转换器的单片机来处理。我们的这款控制板就具有内部 ADC 转换器，当模拟信号输入到控制板中，转变成数字信号后，进行数值分析处理计算出电压值。

5.3 程序讲解

我们首先看怎么获取电池电压，该函数依旧在 App.cpp 内

```

40 uint16 GetADCResult(void)
41 {
42     return analogRead(ADC_BAT);
43 }
44
45 void CheckBatteryVoltage(void)
46 {
47     uint8 i;
48     uint32 v = 0;
49     for(i = 0; i < 8; i++)
50     {
51         v += GetADCResult();
52     }
53     v >>= 3;
54
55     v = v * 1875 / 128; // adc / 1024 * 5000 * 3 (3代表放大3倍, 因为采集电压时电阻分压了)
56     BatteryVoltage = v;
57 }
58
59 uint16 GetBatteryVoltage(void)
60 { // 电压毫伏
61     return BatteryVoltage;
62 }
63

```

图 5.1 获取电池电压函数

首先，我们通过读取电池电压的 AD 检测通道，即 ADC_BAT 这个引脚的模拟值，然后，再用 for 循环取 8 次的采样值，接着把这个值右移三位，即除以 8 取平均值。最后把模拟值转换为电压，通过 GetBatteryVoltage 函数返回电压值。

GetBatteryVoltage 函数在定时器 3 中断中被调用，如下图所示


```

109 ISR(TIMER2_OVF_vect)
110 {
111     TCNT2=206; //定时器3中断 100us
112
113     static uint16 time = 0;
114     static uint16 timeBattery = 0;
115
116     Buzzer();
117     if(++time >= 10)
118     {
119         time = 0;
120         gSystemTickCount++;
121         // Ps2TimeCount++;
122         if(GetBatteryVoltage() < 6400)//小于6.4V报警
123         {
124             timeBattery++;
125             if(timeBattery > 5000)//持续5秒
126             {
127                 BuzzerState = 1;
128             }
129         }
130         else
131         {
132             timeBattery = 0;
133             BuzzerState = 0;
134         }
135     }
136 }

```

图 5.2 定时器 3 的中断

如果电池电压小于 6.4v，则 5 秒后再次检测，如果还是小于 6.4，那么将 BuzzerState 置 1，这个变量我们在上一节讲过，它 1 时会让蜂鸣器响起来，所以电池电压低于 6.4 时，蜂鸣器会响起来。

6 多路舵机控制

6.1 实验器材

控制板、电池、舵机

6.2 基础知识

6.2.1 舵机内部结构

舵机内部包括了一个小型直流马达；一组变速齿轮组；一个线性反馈电位器；及一块控制电路板。其中，高速转动的直流马达提供了舵机的原始动力，带动减速齿轮组，使之产生高扭力的输出，齿轮组的变速比愈大，舵机的输出扭力也愈大，也就是说越能带动更大重量的负载（受齿轮强度限制），但输出的转速（响应速度）也愈低。

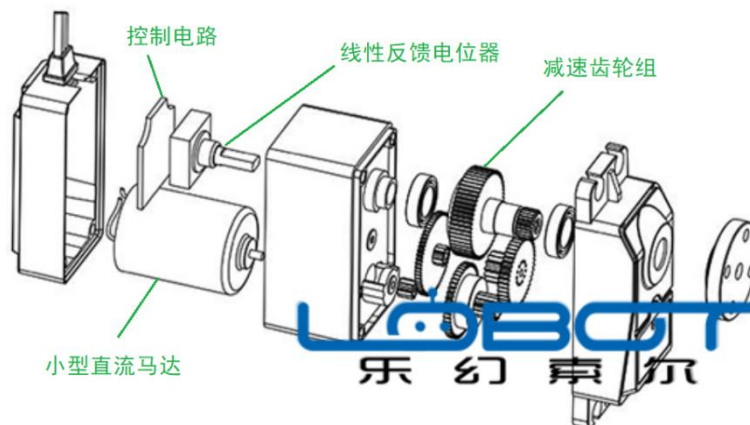


图 6.1 舵机结构图

6.2.2 舵机的工作原理

舵机是一个典型闭环反馈系统，其原理可由下图表示：

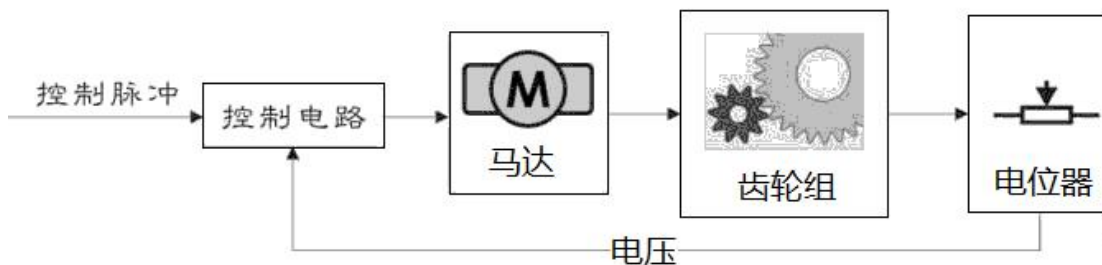


图 6.2 舵机闭环反馈系统

减速齿轮组由马达驱动，其输出端带动一个线性的比例电位器作位置检测，该电位器把转角角度转换为一比例电压反馈给控制电路，控制电路将其与输入的控制信号对应的角度作比较，并驱动马达正向或反向地转动，使电位器反馈角度趋向于控制信号期望角度，从而达到使伺服马达精确定位的目的。

6.2.3 如何控制舵机

标准的 PWM 舵机有三条控制线，分别为：电源、地及控制。

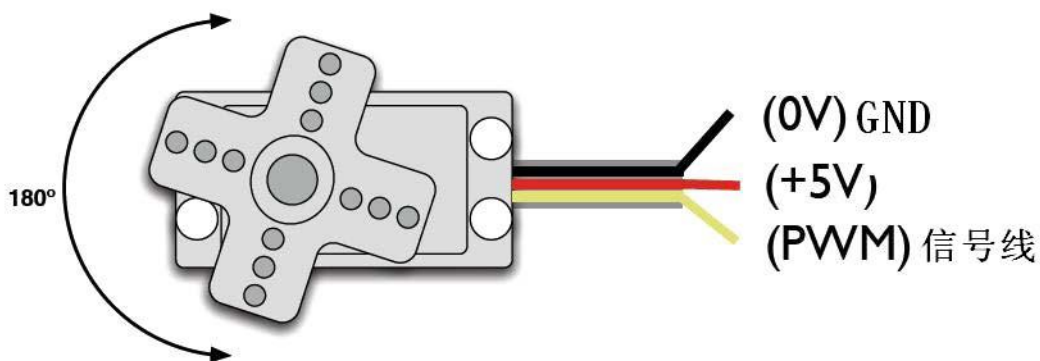


图 6.3 舵机引线

电源线与地线用于提供内部的直流马达及控制线路所需的能源，电压通常介于 5V—8V 之间，该电源应尽可能与处理系统的电源隔离（因为马达会产生噪音）。甚至舵机在重负载时也会拉低放大器的电压，所以整个系统的电源供应的比例必须合理。

输入一个周期性的正向脉冲信号，这个周期性脉冲信号的高电平时间通常在 1ms—2ms 之间，而低电平时间应在 5ms 到 20ms 之间。模拟舵机需要一直保持周期性的信号才可保持舵机的角度，当失去信号，舵机就会不再输出动力。我们使用的是数字舵机，只要发送一次正确的高电平信号就可以保持锁定角度，对低电平时间并不很严格。









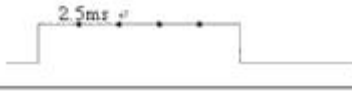

输入正脉冲宽度（周期为 20ms）	舵机输出臂位置
	 $\approx -90^\circ$
	 $\approx -45^\circ$
	 $\approx 0^\circ$
	 $\approx 45^\circ$
	 $\approx 90^\circ$

图 6.4 舵机信号与对应角度

6.2.4 舵机运动速度

舵机的瞬时运动速度是由其内部的直流马达和变速齿轮组的配合决定的，在恒定的电压驱动下，其数值唯一。对于数字 PWM 舵机，其速度由其内部程序确定，一般但其平均运动速度可通过分段停顿的控制方式来改变，例如，我们可把动作幅度为 90° 的转动细分为 128 个停顿点，通过控制每个停顿点的时间长短来实现 0° — 90° 变化的平均速度。对于多数舵机来说，速度的单位由“度数/秒”来决定。

6.3 程序讲解

```

3 void setup()
4 {
5     pinMode(LED, OUTPUT);
6     pinMode(BUZZER, OUTPUT);
7     pinMode(KEY, INPUT_PULLUP);
8
9     InitPWM();
10    InitTimer2();
11
12    InitUart1();
13    InitPS2();
14    InitFlash();
15    InitMemory();
16
17 }
18

```

图 6.1 初始化

初始化舵机连接,让 6 个舵机分别和不同引脚连接,同时设置角度范围限制。

```

88
89 void InitPWM(void)
90 {
91     myservo[0].attach(2, 500, 2500); // attaches the servo on pin 2 to the servo object
92     myservo[1].attach(3, 500, 2500);
93     myservo[2].attach(4, 500, 2500);
94     myservo[3].attach(5, 500, 2500);
95     myservo[4].attach(6, 500, 2500);
96     myservo[5].attach(7, 500, 2500);
97 }

```

图 6.2 舵机初始化连接

```
138 void TaskTimeHandle(void)
139 {
140     static uint32 time = 10;
141     static uint32 times = 0;
142     if(gSystemTickCount > time)
143     {
144         time += 10;
145         times++;
146         if(times % 2 == 0)//20ms
147         {
148             ServoPwmDutyCompare();
149         }
150     }
151 }
152
153
```

图 6.3 舵机运行处理函数

loop 里循环调用 TaskRun，TaskRun 里调用 TaskTimeHandle 这个函数，
此函数会定时调用 ServoPwmDutyCompare 函数

```

30 void ServoPwmDutyCompare(void)//脉宽变化比较及速度控制
31 {
32     uint8 i;
33
34     static uint16 ServoPwmDutyIncTimes; //需要递增的次数
35     static bool ServoRunning = FALSE; //舵机正在以指定速度运动到指定的脉宽对应的位置
36     if(ServoPwmDutyHaveChange)//停止运动并且脉宽发生变化时才进行计算      ServoRunning == FALSE &&
37     {
38         ServoPwmDutyHaveChange = FALSE;
39         ServoPwmDutyIncTimes = ServoTime/20; //当每20ms调用一次ServoPwmDutyCompare()函数时用此句
40         for(i=0;i<8;i++)
41         {
42             //if(ServoPwmDuty[i] != ServoPwmDutySet[i])
43             {
44                 if(ServoPwmDutySet[i] > ServoPwmDuty[i])
45                 {
46                     ServoPwmDutyInc[i] = ServoPwmDutySet[i] - ServoPwmDuty[i];
47                     ServoPwmDutyInc[i] = -ServoPwmDutyInc[i];
48                 }
49                 else
50                 {
51                     ServoPwmDutyInc[i] = ServoPwmDuty[i] - ServoPwmDutySet[i];
52                 }
53             }
54             ServoPwmDutyInc[i] /= ServoPwmDutyIncTimes;//每次递增的脉宽
55         }

```



```

56     }
57     ServoRunning = TRUE; //舵机开始动作
58 }
59 if(ServoRunning)
60 {
61     ServoPwmDutyIncTimes--;
62     for(i=0;i<8;i++)
63     {
64         if(ServoPwmDutyIncTimes == 0)
65         { //最后一次递增就直接将设定值赋给当前值
66
67             ServoPwmDuty[i] = ServoPwmDutySet[i];
68
69             ServoRunning = FALSE; //到达设定位置，舵机停止运动
70         }
71         else
72         {
73
74             ServoPwmDuty[i] = ServoPwmDutySet[i] +
75                 (signed short int)(ServoPwmDutyInc[i] * ServoPwmDutyIncTimes);
76
77         }
78         if((i >= 0) && (i <= 6))
79         {
80             myservo[i - 1].writeMicroseconds(ServoPwmDuty[i]);
81         }
82

```

图 6.4 ServoPwmDutyCompare 函数

当我们设定下图中函数的参数值，也就是舵机转动的指令时，ServoPwmDutyCompare 函数就会根据设定的值，对时间进行分解，然后循环判断所有舵机需要变动的角度，接着通过分解的时间让舵机分成多次转动，从而实现对时间的控制，也就是对舵机速度的控制，时间越短，速度越快，舵机转动用 writeMicroseconds 这个指令。

我们用下面这个含有三个参数的函数来驱动每个舵机，舵机 id 号限制为 0-7，位置 500-2500（0-180），时间 20ms-30000ms。


```

14 void ServoSetPluseAndTime(uint8 id, uint16 p, uint16 time)
15 {
16     if(id >= 0 && id <= 7 && p >= 500 && p <= 2500)
17     {
18         if(time < 20)
19             time = 20;
20         if(time > 30000)
21             time = 30000;
22         ServoPwmDutySet[id] = p;
23         ServoTime = time;
24         ServoPwmDutyHaveChange = TRUE;
25     }
26
27 }

```

图 6.5 舵机转动指令

7 PS2 手柄控制舵机运动

7.1 实验器材

PS 手柄、接收器、控制板、舵机、电池

7.2 基础知识

人机交互界面在控制系统中十分重要，手柄的操作方便直观，非常适合在机器人控制中使用。此处，我们选用常见的 PS 游戏手柄作为操纵设备。PS 手柄的与单片机通信时只需要 4 根信号线。手柄与到单片机通过串行方式通讯，占用的 IO 口较少且通讯协议比较简单，所以非常适合我们的需要。

下面是 PS 手柄接收器的引脚定义如下图

引脚	定义	用途
1	DATA	从手柄到主机的串行数据线，此信号是一个8bit的串行数据，同步传送于时钟下降沿（输入输出信号在时钟信号由高到低时变化，所有信号的读取在时钟前沿到电平变化前完成。）
2	COMMAND	从主机到手柄的串行数据线，工作方式于DATA信号相同
3	NC	未使用
4	GND	电源地和信号地
5	3.3v	电源电压，有效工作电压3V-5V
6	Attention	用于提供手柄触发信号，信号在通信期间处于低电平。相当于片选信号
7	CLOCK	信号方向：从主机到手柄，用于保持数据同步
8	NC	未用
9	acknowledge	从手柄到主机的应答信号，此信号在每个8Bit数据发送之后的最后一个时钟周期变低，并且ATT一直保持低电平，如果ACK信号不变低约60us，PS主机试另外的手柄

图 7.1 PS 手柄接收器引脚说明

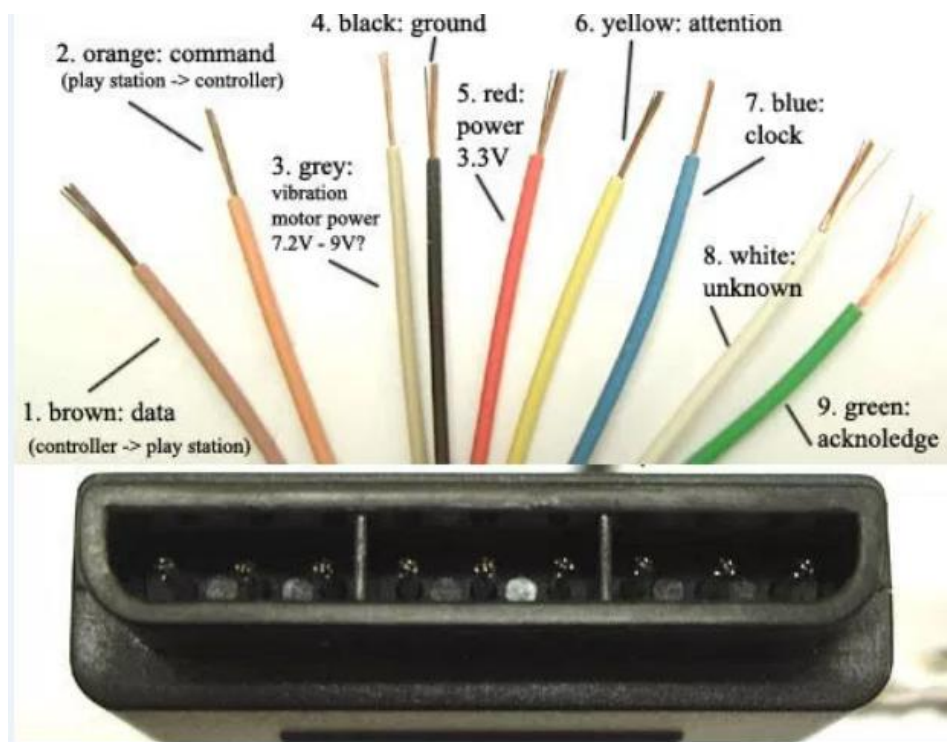


图 7.2 PS 手柄接收器实物图

7.3 程序讲解

loop 中循环 TaskRun() , TaskRun()中调用 了 ps2Handle 这个函数。

ps2 的初始化在 setup 部分的 InitPS2 函数中。

```

14 PS2X ps2X; //实例化手柄类
15 void InitPS2()
16 {
17     ps2X.config_gamepad(A2, A4, A3, A5); //设置PS2接口 A2号IO为clock, A4号IO为command, A3号IO为attention, A5号IO为data
18 }

```

图 7.3 ps2 初始化

然后，ps2Handle 函数截取部分讲解，剩下的部分类似

```

154 void ps2Handle() { //PS2 手柄 处理
155     static uint32_t Timer; //定义静态变量Timer, 用于计时
156     if (Timer > millis()) //Timer 大于 millis() (运行的总毫秒数)时返回, //Timer 小于 运行总毫秒数时继续运行下面的语句
157         return;
158
159     ps2X.read_gamepad(); //读取PS手柄按键数据
160
161
162     if (ps2X.ButtonPressed(PSB_START)) { //如果左侧向上按钮被按下
163         LedFlip();
164         FullActRun(0,1);
165         Timer = millis() + 50; //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
166         return; //返回, 退出此函数
167     }
168     if (ps2X.ButtonPressed(PSB_PAD_UP)) { //如果左侧向上按钮被按下
169         LedFlip();
170         FullActRun(1,1);
171         Timer = millis() + 50; //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
172         return; //返回, 退出此函数
173     }
174     if (ps2X.ButtonPressed(PSB_PAD_DOWN)) { //如果左侧向下按钮被按下
175         LedFlip();
176         FullActRun(2,1);
177         Timer = millis() + 50; //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
178         return; //返回, 退出此函数

```

图 7.4 部分 ps2Handle 程序

进入 ps2Handle 函数, 首先会延时一段时间再读取数据, 确保数据的读取完毕, 然后根据按键的值执行相应的操作, 其中 LedFlip 为 led 灯闪烁, 用来提示客户按键成功, FullActRun 是动作组运行函数, 在后面的章节会讲

到，它实际上就是调用控制器里下载好的动作组，接着延时，退出。

8 SPI_Flash 读写

1. 实验器材

控制板

2. 基础知识

FLASH 闪存，是一种非易失性存储器，在没有电流供应的情况也能够长久地保存数据，其存储特性相当于硬盘，这项特性正是闪存得以成为各类便携型数字设备的存储介质的基础。

SPI 协议是由摩托罗拉公司提出的通讯协议(Serial Peripheral Interface)，即串行外围设备接口，是一种高速全双工的通信总线。它被广泛地使用在 ADC、LCD 等设备与 MCU 间，要求通讯速率较高的场合。

SPI FLASH 就是一种通过 SPI 接口读写的 FLASH 存储器。一般的 SPI FLASH 的读写有以下两个特性：

- 1、写入的时候只可写入 1，不可以写入 0
- 2、擦除的时候是按扇区擦除(即将所有数据都变成 0)，扇区大小不同芯片有所不同（我们选用的芯片每扇区为 4096 字节）

综合上面两点我们可以知道，一个字节的数据就是将芯片内对应的数据位由 0 变成 1，或者由 1 变成 0。因为写入的时候 FLASH 不支持写入 0，所以我们要将对应扇区擦除才可变成 0。但是，擦除后原本的数据就会丢失，所以一般是先读出该扇区数据后擦除该扇区再将修改后的数据重新写入 FLASH。

3. 程序讲解

InitFlash 函数在 setup 部分被调用，它的函数本体在 Flash.cpp 中，它调用了 InitSpi 函数，此函数是对 SPI 的初始化。

```

4  //*****
5  SPI初始化
6  入口参数：无
7  出口参数：无
8  *****/
9  void InitSpi(void)
10 {
11     pinMode(SS, OUTPUT);
12
13     SPI.begin (); //初始化SPI总线，将SCK(Pin13)，MOSI(Pin11)和SS(Pin10)管脚设置为输出模式，将SCK和MOSI设置为低电平，SS为高电平。
14     SPI.setDataMode(SPI_MODE0); //SPI_MODE0（上升沿采样，下降沿置位，SCK闲置时为0），
15                                     SPI_MODE1（上升沿置位，下降沿采样，SCK闲置时为0），
16                                     SPI_MODE2（下降沿采样，上升沿置位，SCK闲置时为1），
17                                     SPI_MODE3（下降沿置位，上升沿采样，SCK闲置时为1）。*/
18     SPI.setBitOrder(MSBFIRST); //设置串行数据传输时，先传输高位还是低位，有LSBFIRST（最低位在前）和MSBFIRST（最高位在前）两种类型可选。
19 }

```

图 8.1 初始化 SPI

读写 SPI Flash 需要向 SPI Flash 发送相应的命令，这些命令通常都在芯片手册上说明。

下图为 SPI FLASH 的命令表：

INSTRUCTION NAME	BYTE 1 (CODE)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Write Enable	06h					
Write Disable	04h					
Read Status Register-1	05h	(S7–S0) ⁽²⁾				
Read Status Register-2	35h	(S15-S8) ⁽²⁾				
Write Status Register	01h	(S7–S0)	(S15-S8)			
Page Program	02h	A23–A16	A15–A8	A7–A0	(D7–D0)	
Quad Page Program	32h	A23–A16	A15–A8	A7–A0	(D7–D0, ...) ⁽³⁾	
Block Erase (64KB)	D8h	A23–A16	A15–A8	A7–A0		
Block Erase (32KB)	52h	A23–A16	A15–A8	A7–A0		
Sector Erase (4KB)	20h	A23–A16	A15–A8	A7–A0		
Chip Erase	C7h/60h					
Erase Suspend	75h					
Erase Resume	7Ah					
Power-down	B9h					
High Performance Mode	A3h	dummy	dummy	dummy		
Mode Bit Reset ⁽⁴⁾	FFh	FFh				
Release Power down or HPM / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0) ⁽⁵⁾	
Manufacturer/ Device ID ⁽⁶⁾	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)
Read Unique ID ⁽⁷⁾	4Bh	dummy	dummy	dummy	Dummy	(ID63-ID0)
JEDEC ID	9Fh	(M7-M0) Manufacturer	(ID15-ID8) Memory Type	(ID7-ID0) Capacity		

图 8.2 SPI FLASH 命令表

我们根据这个表格将各个可能用到的命令 用宏定义在头文件中定义出来，

方便使用

```

0
7 #define SFC_WREN          0x06          // 串行Flash命令集
8 #define SFC_WRDI          0x04
9 #define SFC_RDSR          0x05
10 #define SFC_WRSR          0x01
11 #define SFC_READ          0x03
12 #define SFC_FASTREAD      0x0B
13 #define SFC_RDID          0xAB
14 #define SFC_PAGEPROG      0x02
15 #define SFC_RDCR          0xA1
16 #define SFC_WRCR          0xF1
17 #define SFC_SECTORER      0xD7
18 #define SFC_BLOCKER       0xD8
19 #define SFC_SECTOR_ERASE  0x20
20 #define SFC_CHIPER        0xC7
21

```

图 8.3 Flash 命令表

下面实现了检测 Flash 是否处于忙状态。Flash 处于忙状态的时候不会接收我们的读写命令。所以在读写前我们要先检测 Flash 的忙状态知道 Flash 空闲后才可发起读写操作。

```

26
27 void CheckBusy()
28 {
29     digitalWrite(SS, HIGH);
30     digitalWrite(SS, LOW);
31     SPI.transfer(SFC_RDSR); //transfer: 用于在SPI总线上传输一个数据，包括发送与接收。
32     while(SPI.transfer(0) & 0x01); //SPI.transfer(0)读取状态，返回1表示忙，0表示空闲
33     digitalWrite(SS, HIGH);
34 }
35

```

图 8.4 Flash 忙检测

```

37 void FlashRead(DWORD addr, DWORD size, BYTE *buffer)
38 {
39     CheckBusy(); //忙检测，确保空闲
40     digitalWrite(SS, LOW); //拉低选中
41     SPI.transfer(SFC_READ); //发送快速读取命令
42     SPI.transfer((BYTE)(addr>>16)); //高八位
43     SPI.transfer((BYTE)(addr>>8)); //中八位
44     SPI.transfer((BYTE)addr); //低八位
45     for(int i = 0; i < size; i++)
46     {
47         buffer[i] = SPI.transfer(0); //第五个字节开始为flash返回的内部储存数据，括号中的0可用任意值，只是用来触发sck
48     }
49     digitalWrite(SS, HIGH); //拉高关闭传输
50 }
51
52 void FlashWrite(DWORD addr, DWORD size, BYTE *buffer)
53 {
54     digitalWrite(SS, HIGH);
55     digitalWrite(SS, LOW);
56     SPI.transfer(SFC_WREN); //发送写使能命令
57     digitalWrite(SS, HIGH);
58     digitalWrite(SS, LOW);
59     SPI.transfer(SFC_PAGEPROG); //发送页编辑命令
60     SPI.transfer((BYTE)(addr>>16)); //设置起始地址
61     SPI.transfer((BYTE)(addr>>8));
62     SPI.transfer((BYTE)addr);
63     for(int i = 0; i < size; i++)

```

图 8.5 读写函数

读取之前先进行忙检测，然后拉低选中，接着使用快速读取命令，然后发送需要读取的地址，最后用一个 for 循环把读到的数据储存到 buffer 中。写的操作和读差不多，只不过忙检测放在最后。

最后，擦除 Flash 扇区的指令如下。

```
72 void FlashEraseSector(DWORD addr)
73 {
74     digitalWrite(SS, HIGH);
75     digitalWrite(SS, LOW);
76     SPI.transfer(SFC_WREN); //使能Flash写命令
77     digitalWrite(SS, HIGH);
78     digitalWrite(SS, LOW);
79     SPI.transfer(SFC_SECTOR_ERASE); //擦除指令
80     SPI.transfer((BYTE) (addr>>16));
81     SPI.transfer((BYTE) (addr>>8));
82     SPI.transfer((BYTE) addr);
83     digitalWrite(SS, HIGH);
84     CheckBusy();
85 }
86
```

图 8.6 擦除扇区

9 通过串口控制舵机转动

9.1 实验器材

控制板、舵机、电池、数据线

9.2 基础知识

串行通讯端口,简称 串口,是指串行式逐位传输的通讯接口。在单片机、嵌入式环境中所说串口一般特指 UART 口。按接口的电平标准分为 RS-232、RS-422、RS485、TTL 等。

通常我们从单片机芯片引出后没有经过专用芯片转换的都是 TTL 串口。两种物理接口提供的。一是 D 形状 9 针插头(DB9)和 4 针排针两种。

9.3 程序讲解

首先,我们看在 setup 部分的串口初始化 InitUart1 函数。

```
11 void InitUart1(void)
12 {
13     //bitSet(UCSR0A,U2X0);
14     bitSet(UCSR0B,RXCIE0); //将USART0 I/O 数据寄存器的RXCIE0位置1, 接收结束中断使能
15     bitSet(UCSR0B,RXEN0); //bitSet (value, bit) = ((value) |= (1UL << (bit))) 接收使能
16     bitSet(UCSR0B, TXEN0); //发送使能
17     bitSet(UCSR0C, UCSZ01); //将USART1波特率寄存器的UCSZ01位置1
18     bitSet(UCSR0C, UCSZ00); //设置数据帧包含的数据位数, 8位
19     UBRR0=(F_CPU/16/9600-1); //波特率9600 F_CPU系统频率 U2X0 = 0 , 异步正常工作模式, /16
20 }
```

图 9.1 InitUart1 函数

此函数是对串口进行初始化, 通过对寄存器的一系列操作把它设置成可传输数据, 波特率为 9600。

接着我们看在 loop 中调用的 TaskPCMsgHandle 函数。

```

145 void TaskPCMsgHandle(void)
146 {
147
148     uint16 i;
149     uint8 cmd;
150     uint8 id;
151     uint8 servoCount;
152     uint16 time;
153     uint16 pos;
154     uint16 times;
155     uint8 fullActNum;
156     if(UartRxOK())
157     {
158         // LED = !LED;
159         if(digitalRead(LED) == HIGH)
160         {
161             digitalWrite(LED, LOW);
162         }
163         else
164         {
165             digitalWrite(LED, HIGH);
166         }
167
168         cmd = UartRxBuffer[3];
169         switch(cmd)
170         {
171             case CMD_MULT_SERVO_MOVE:
172                 servoCount = UartRxBuffer[4];

```

图 9.2 TaskPCMsgHandle 函数

先对 UartRxOK 函数的返回值进行判断，UartRxOK 函数我们往上拉，发现它要对 fUartRxComplete 值先做判断，然后我们发现 fUartRxComplete 值发生变化实在 ISR 函数里，而 ISR(USART_RX_vect)函数是个中断函数。

```

39 ISR(USART_RX_vect) //数据从移位寄存器完整移动到接收寄存器, USART的3个中断之一
40 {
41     uint8 i;
42     uint8 rxBuf;
43
44     static uint8 startCodeSum = 0;
45     static bool fFrameStart = FALSE;
46     static uint8 messageLength = 0;
47     static uint8 messageLengthSum = 2;
48
49     rxBuf=UDR0;
50     if(!fFrameStart)
51     {
52         if(rxBuf == 0x55)
53         {
54             startCodeSum++;
55             if(startCodeSum == 2)
56             {
57                 startCodeSum = 0;
58                 fFrameStart = TRUE;
59                 messageLength = 1;
60             }
61         }
62         else
63         {
64
65             fFrameStart = FALSE;
66             messageLength = 0;
67
68             startCodeSum = 0;
69         }
70     }
71
72     if(fFrameStart)
73     {
74         Uart1RxBuffer[messageLength] = rxBuf;
75         if(messageLength == 2)
76         {
77             messageLengthSum = Uart1RxBuffer[messageLength];
78             if(messageLengthSum < 2)// || messageLengthSum > 30
79             {
80                 messageLengthSum = 2;
81                 fFrameStart = FALSE;
82             }
83         }
84     }
85
86     messageLength++;
87
88     if(messageLength == messageLengthSum + 2)
89     {
90         if(fUartRxComplete == FALSE)
91     {

```

图 9.3 ISR 函数

对这段程序我们可以假设发送的数据帧是 “0x55 0x55 0x05 0x03 0x01 0xD0 0X07” 这条信息是指控制 1 号舵机在 1000ms 内转到 2000 的位置，我们结合程序来说明，首先，当接收到数据时，将数据储存在 rxBuf 中，如果接收到数据帧 0x55 则 startCodeSum+1，当接收到两个 0x55 时 startCodeSum 清零，fFrameStart 置为 ture 则下一帧数据开始保存，如果没有接收到数据则重新判定。fFrameStart 为 ture 即接收了两个 0x55 后，将 0x55 保存到数组 Uart1RxBuffer 中的第二个位置（下标为 1），当接收到第三个数据即 0x05（此帧表示数据长度，数据长度为数据帧除两个帧头外的部分的字节数，即数据帧总长度-2）时将它保存在 messageLengthSum 中，然后判断它的大小，因为一个完整的数据至少需要两个帧头 0x55 和一个数据长度，一个指令，即数据长度为 2，所以如果数据长度小于 2，那么数据传输就出现问题，则重新开始；如果没有问题，则继续下一帧数据的接收，直到接收的总帧长度（我们发的这个数据总帧长度为 7）等于数据长度（5）加 2，即接收到一个完整的数据。

然后我们回去看 TaskPCMsgHandle 函数，此时 UartRxOK 会返回 ture。然后，我们将接收到的数据按指令值分类判断，将其他数据值转化为舵机 ID，转动时间、位置。最后通过 ServoSetPluseAndTime（此函数在第 6 节已介绍过）这个函数实现舵机转动。

10 存储动作组到 Flash 中

10.1 实验器材

控制板

10.2 基础知识

前面我们已经完成了 SPI Flash 的读写,下面我们要将动作数据保存到 Flash 中。它简单的工作过程如下:

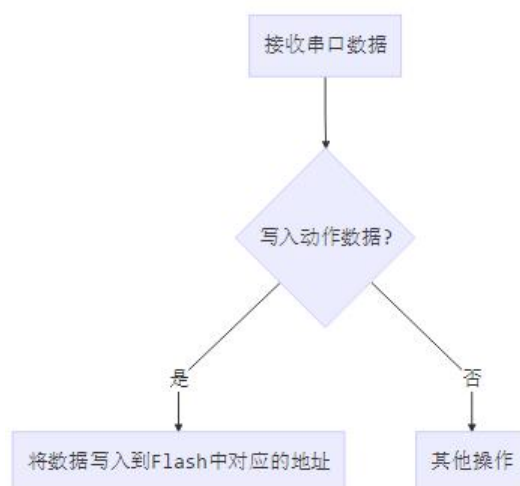


图 10.1 FLASH 读写操作流程

从上可知,我们需要从串口接收到的数据中提取出动作数据,然后正确的写入 Flash 中。

上一节中我们已经知道了 Flash 擦除是以扇区为单位的,同时只可以写入 1 不能写入 0。所以为了方便编程、管理,我们在使用 FLASH 的时候将存储地址以我们所用的 Flash 芯片的扇区大小 4096 字节对齐,这样我们只要写入我们的目标数据即可,不用全部写入 4096 个字节,提高效率减少出错的概率。

10.3 程序讲解

```

192
193     case CMD_FULL_ACTION_ERASE:
194         FlashEraseAll();
195         McuToPCSendData(CMD_FULL_ACTION_ERASE, 0, 0);
196         break;
197
198     case CMD_ACTION_DOWNLOAD:
199         SaveAct(UartRxBuffer[4], UartRxBuffer[5], UartRxBuffer[6], UartRxBuffer + 7);
200         McuToPCSendData(CMD_ACTION_DOWNLOAD, 0, 0);
201         break;

```

图 10.2 动作组擦除与下载

上面程序我们可以看到，我们增加了擦除动作组和动作下载的功能。两个功能分别是两个函数实现 FlashEraseAll()和 SaveAct(...)。

两个命令的协议如下：

擦除动作组

指令名 **CMD_FULL_ACTION_ERASE** 指令值 8 数据长度：3

说明：将下载到控制板的动作组擦除

参数 1：（保留）

返回：控制板返回不带参数的指令

下载动作指令

指令名 **CMD_ACTION_DOWNLOAD** 指令值 25 数据 长度 N：

说明：通过串口下载动作组，一帧一帧的下载，该动作组有多少帧就会下载多少次。

数据长度 $N = \text{下载舵机的个数} \times 3 + 8$

参数 1：要下载到的动作组编号

参数 2：该动作组的总帧数

参数 3：第几帧数据

参数 4：要下载舵机的个数

参数 5：时间低八位

参数 6：时间高八位

参数 7：舵机 ID 号

参数 8：角度位置低八位

参数 9：角度位置高八位 参数.....：格式与参数 7,8,9 相同，不同 ID 的角度位置。每下载一帧数据，板子都会返回数据，返回数据的指令值是相同的，不过是不带参数的指令包。

在实现这两个函数之前我们要先安排好数据在 Flash 中的存放位置。下图就是我们的数据存放的地址分布：



图 10.3 数据存放地址

第一个扇区存放 LOBOT 标志, 如果没有 LOBOT 标志就认为这是一块新的 Flash 芯片。

第二扇区存放每个动作组的动作个数, 每个动作组占用一个字节, 即每个动作组可以存放 255 个动作。我们只使用这个扇区的前 256 个字节, 就是最多一共存储 256 个动作组。操作的时候要全部读出 256 个字节的数据, 然后擦除这个扇区在将修改后的数据重新写入

从第三个扇区开始就是存放我们动作组数据区域。每个动作组占用 16KB, 就是 4 个扇区。

写入一个动作组的过程如下：

第一步，擦除要写入的动作组的存储区域

第二步，依次写入动作组的动作，同时检查写入的动作是不是这个动作组的最后一个动作

第三步，如果是最后一个动作我们就认为动作写入完毕，更新第二扇区的动作组个数为新的动作个数

擦除一个动作的方法就是将对对应动作组的动作个数改为 0。而擦除全部动作组的方法就是直接将第二扇区的前 256 个字节都改为 0 即可。

下面是此部分程序的实现

```
205 void SaveAct(uint8 fullActNum, uint8 frameIndexSum, uint8 frameIndex, uint8* pBuffer)
206 {
207     uint8 i;
208
209
210     if(frameIndex == 0)//下载之前先把这个动作组擦除
211     { //一个动作组占16k大小，擦除一个扇区是4k，所以要擦4次
212
213         for(i = 0; i < 4; i++)//ACT_SUB_FRAME_SIZE/4096 = 4
214         {
215             FlashEraseSector((MEM_ACT_FULL_BASE) + (fullActNum * ACT_FULL_SIZE) + (i * 4096));
216         }
217     }
218
219     FlashWrite((MEM_ACT_FULL_BASE) + (fullActNum * ACT_FULL_SIZE) + (frameIndex * ACT_SUB_FRAME_SIZE)
220         , ACT_SUB_FRAME_SIZE, pBuffer);
221
222     if((frameIndex + 1) == frameIndexSum)
223     {
224         FlashRead(MEM_FRAME_INDEX_SUM_BASE, 256, frameIndexSumSum);
225         frameIndexSumSum[fullActNum] = frameIndexSum;
226         FlashEraseSector(MEM_FRAME_INDEX_SUM_BASE);
227         FlashWrite(MEM_FRAME_INDEX_SUM_BASE, 256, frameIndexSumSum);
228     }
229 }
```

图 10.4 动作组下载

```

231
232 void FlashEraseAll()
233 {
234     uint16 i;
235
236     for(i = 0; i <= 255; i++)
237     {
238         frameIndexSumSum[i] = 0;
239     }
240     FlashEraseSector(MEM_FRAME_INDEX_SUM_BASE);
241     FlashWrite(MEM_FRAME_INDEX_SUM_BASE, 256, frameIndexSumSum);
242 }
243

```

图 10.5 动作组擦除

```

---
244 void InitMemory(void)
245 {
246     uint8 i;
247     uint8 logo[] = "LOBOT";
248     uint8 datatemp[8];
249
250     uint8 tt[4] = {0, 1, 2, 3};
251     uint8 ttt[4] = {5, 5, 5, 5};
252
253     FlashRead(MEM_LOBOT_LOGO_BASE, 5, datatemp);
254     for(i = 0; i < 5; i++)
255     {
256         if(logo[i] != datatemp[i])
257         {
258             FlashEraseSector(MEM_LOBOT_LOGO_BASE);
259             FlashWrite(MEM_LOBOT_LOGO_BASE, 5, logo);
260             FlashEraseAll();
261             break;
262         }
263     }
264 }
265

```

图 10.6 判断是否新区域

11 运行存储在 Flash 中的动作组（完整的机械臂程序）

12.1 实验器材

控制板

12.2 基础知识

本节我们将实现通过串口命令控制机械臂运行指定的动作组。我们需要实现运行动作组的功能并且实现对应的串口命令和 PS2 手柄按键调用对应动作组。

我们设计好对应的串口指令协议如下；

运行动作组指令

指令名 **CMD_ACTION_RUN** 指令值 6 数据 长度 5

说明：动作组运行，如果参数次数为无限次，则此参数值为 0

参数 1：要运行的动作组的编号参数

参数 2：动作组要执行的次数低八位参数

参数 3：动作组要执行的次数高八位

我们要先实现运行 SPI Flash 中的动作组的功能。

动作组是有一个个动作的组合，动作其实就是舵机位置的变化。前面章节中我们已经实现了控制多路舵机的速度动作，那现在我们只要以合适的时间来改变这些舵机的角度就可以实现一个动作。而一个个动作串接起来就是一个动作组。

12.3 程序讲解

要运行动作组我们要先设置好运行动作组的环境，例如动作组内是否由动作，包含的动作个数，检查当前有没有动作组在运行。然后根据不同的情况做对应处理，以防止出错。然后设定好运行动作组的参数，设置运行动作组的标志。

```

16 void FullActRun(uint32 actFullnum, uint32 times) //初始化并运行新的动作
17 {
18     uint8 frameIndexSum;
19     FlashRead(MEM_FRAME_INDEX_SUM_BASE + actFullnum, 1, &frameIndexSum);
20     UART1SendDataPacket(&frameIndexSum, 1);
21     if(frameIndexSum > 0) //该动作组的动作数大于0，说明是有效的，已经下载过动作了。
22     {
23         FrameIndexSum = frameIndexSum;
24         if(ActFullNum != actFullnum)
25         {
26             if(actFullnum == 0)
27             { //0号动作组强制运行，可以中断当前正在运行的其他动作组
28                 fRobotRun = FALSE;
29                 ActFullRunTimes = 0;
30                 fFrameRunFinish = TRUE;
31             }
32         }
33         else
34         { //只用前后两次动作组编号相同才能修改次数
35             ActFullRunTimesSum = times;
36         }
37
38
39         if(FALSE == fRobotRun)
40         {
41             ActFullNum = actFullnum;
42             ActFullRunTimesSum = times;

```

图 11.1 初始化

我们先实现运行一个动作。运行一个动作就是从 Flash 中对应的地方读出这个动作的运行时间和各个舵机的角度,然后让我们前面已经写好的控制舵机的程序来控制舵机以指定时间转动对应角度。

```
65 uint16 ActSubFrameRun(uint8 fullActNum, uint8 frameIndex)
66 {
67     uint32 i = 0;
68
69     // uint16 frameSumSum = 0; //由于子动作都是连续存放的,子动作的帧数又是不确定的数
70     //,所以要总在一起算。所有前面子动作的帧加起来
71     uint8 frame[ACT_SUB_FRAME_SIZE];
72     uint8 servoCount;
73     uint32 time;
74     uint8 id;
75     uint16 pos;
76
77     FlashRead((MEM_ACT_FULL_BASE) + (fullActNum * ACT_FULL_SIZE) + (frameIndex * ACT_SUB_FRAME_SIZE)
78         , ACT_SUB_FRAME_SIZE, frame);
79
80     servoCount = frame[0];
81     time = frame[1] + (frame[2]<<8);
82
83     if(servoCount > 8)
84     { //舵机数超过8个,说明下载了错误动作
85         FullActStop();
86         return 0;
87     }
88     for(i = 0; i < servoCount; i++)
89     {
90         id = frame[3 + i * 3];
91
92         pos = frame[4 + i * 3] + (frame[5 + i * 3]<<8);
93         ServoSetPluseAndTime(id, pos, time);
94     }
95     return time;
96 }
```

图 11.2 运行一个动作组

我们看到运行动作的实现其实就是从 Flash 中对应的位置读出动作数据,然后调用 ServoSetPluseAndTime 函数。中间加了一些判断数据是否正确的处理,防止数据出错后的错误动作。

要执行完一个动作组，我们就需要能够自动执行完一个动作后执行下一个动作直至动作组内的动作全部执行完。我们要让程序在执行了一个动作后延时这个动作的执行时间，当执行时间过了之后我们就认为这个动作执行完了，就执行新的动作。同时，这个函数检查动作组运行标志，标志为真就执行动作组，假就不执行。

```
98 void TaskRobotRun(void)
99 {
100
101     if(fRobotRun)
102     {
103         if(TRUE == fFrameRunFinish)
104             //运行完成后开始下一帧动作运行
105             fFrameRunFinish = FALSE;
106         TimeActionRunTotal += ActSubFrameRun(ActFullNum, FrameIndex); //将这帧动作的时间累加上
107     }
108     else
109     {
110         if(gSystemTickCount >= TimeActionRunTotal)
111             //不断检测这帧动作在指定时间内运行完成
112             fFrameRunFinish = TRUE;
113         if(++FrameIndex >= FrameIndexSum)
114             //已运行完该动作组最后一个动作
115             FrameIndex = 0;
116         if(ActFullRunTimesSum != 0)
117             //如果运行次数等于0，即代表无限次运行，就不进入if语句，就一直运行了
118             if(++ActFullRunTimes >= ActFullRunTimesSum)
119                 //到达运行次数，运行停止
120                 fRobotRun = FALSE;
121     }
122 }
123
124 }
```

图 11.3 连续运行多个动作

`gSystemTickCount` 返回程序开始运行到此刻所经过的毫秒数。此刻毫秒数加上动作运行的时间，就是到动作运行完成的时候整个程序经过的毫秒数。当这

个毫秒数到达我们就认为动作运行完成可以运行下个动作或者整个动作组已经运行完毕。

实现了运行动作组的功能后，我们再在串口数据处理函数中增加处理运动动作组命令的代码。

修改后的部分程序代码如下：

```

182
183     case CMD_FULL_ACTION_RUN:
184         fullActNum = UartRxBuffer[4]; //动作组编号
185         times = UartRxBuffer[5] + (UartRxBuffer[6]<<8); //运行次数
186         FullActRun(fullActNum, times);
187         break;
188
189     case CMD_FULL_ACTION_STOP:
190         FullActStop();
191         break;
192

```

图 11.4 新增指令内容

停止动作组就是调用停止动作组函数，这个函数就是设置几个变量和标志，这样 TaskRobotRun 函数就不会再运行动作组了。

```

54 void FullActStop(void)
55 {
56     fRobotRun = FALSE;
57     ActFullRunTimes = 0;
58
59     fFrameRunFinish = TRUE;
60
61     FrameIndex = 0;
62 }
63

```

图 11.5 停止运行动作组

然后我们在 loop 中读取 PS2 手柄的按钮状态，也根据不同的按钮状态进行不同的操作。例如，当向上按钮被按下就执行 1 动作组，START 按钮被按下就执行 0 号动作组等。

```
162  if (ps2X.ButtonPressed(PSE_START)) { //如果左侧向上按钮被按下
163      LedFlip();
164      FullActRun(0, 1);
165      Timer = millis() + 50;          //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
166      return;          //返回，退出此函数
167  }
168  if (ps2X.ButtonPressed(PSE_PAD_UP)) { //如果左侧向上按钮被按下
169      LedFlip();
170      FullActRun(1, 1);
171      Timer = millis() + 50;          //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
172      return;          //返回，退出此函数
173  }
174  if (ps2X.ButtonPressed(PSE_PAD_DOWN)) { //如果左侧向下按钮被按下
175      LedFlip();
176      FullActRun(2, 1);
177      Timer = millis() + 50;          //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
178      return;          //返回，退出此函数
179  }
180  if (ps2X.ButtonPressed(PSE_PAD_LEFT)) { //如果左侧向左按钮被按下
181      LedFlip();
182      FullActRun(3, 1);
183      Timer = millis() + 50;          //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
184      return;          //返回，退出此函数
185  }
186  if (ps2X.ButtonPressed(PSE_PAD_RIGHT)) { //如果左侧向右按钮被按下
187      LedFlip();
```

图 11.5 按键调用动作组

至此我们的机械臂的功能已经基本实现了。

12 增加单舵机模式

12.1 实验器材

机械臂

12.2 基础知识

前面实现的功能里我们实现了按键调用对应的动作组。但是有时候运行预先编好的动作组并不能满足我们的需要。我们有时候还需要直接控制某一个舵机运动。为了实现这个功能我们增加一个单舵机模式。开机是默认的动作组模式：按键可以运行对应的动作组。当按住 SELECT 后再按下 START 键可切换到单舵机模式。单舵机模式下按不同按键对应不同舵机的正反转。

12.3 程序讲解

```
10 uint8 Mode = 0;
```

图 12.1 模式变量

我们增加一个变量 Mode 用来保存模式，0 为动作组模式，1 为单舵机模式

```

158 void ps2Handle() { //PS2 手柄 处理
159     static uint32_t Timer; //定义静态变量Timer， 用于计时
160
161     if (Timer > millis()) //Timer 大于 millis () （运行的总
162         return;
163
164     ps2X.read_gamepad(); //读取PS手柄按键数据
165
166     if( Mode == 0 )
167     {
168         if( ps2X.Button( PSB_SELECT ) )
169         {
170             if( ps2X.ButtonPressed( PSB_START ) )
171             {
172                 Mode = 1;
173                 manual = TRUE;
174                 BuzzerState = 1;
175                 delay(80);
176                 manual = FALSE;
177                 delay(50);
178                 manual = TRUE;
179                 BuzzerState = 1;
180                 delay(80);
181                 manual = FALSE;
182                 FullActRun(0, 1);
183                 LedFlip();
184             }
185         }
186     }
187     Timer = millis();
188 }

```

图 12.2 模式切换

上面代码我们使用了 ps2X.Button 和 ps2X.ButtonPressed 两个函数。

ps2X.Button 能返回一个按钮是否被按住，而 ps2X.ButtonPressed 能返回一个按钮是否刚被按下。当按住了 SELECT 按钮，然后再按下 START 按钮就进行

模式切换。将 Mode 的值改为对应单舵机模式的 1，蜂鸣器响一次，LED 灯闪烁一次，停止动作组运行，将所有舵机都转到 1500 的位置作为初始位置。

```

178  {
179      if (ps2X.ButtonPressed(PSB_START)) { //如果左侧向上按钮被按下
180          LedFlip();
181          FullActRun(0, 1);
182          Timer = millis() + 50;          //Timer 在 运行总毫秒数上加 50ms, 50ms 后再次运行
183          return;          //返回，退出此函数
184      }
185      if (ps2X.ButtonPressed(PSB_PAD_UP)) { //如果左侧向上按钮被按下
186          LedFlip();
187          FullActRun(1, 1);
188          Timer = millis() + 50;
189          return;
190      }
191      if (ps2X.ButtonPressed(PSB_PAD_DOWN)) { //如果左侧向下按钮被按下
192          LedFlip();
193          FullActRun(2, 1);
194          Timer = millis() + 50;
195          return;
196      }
197      if (ps2X.ButtonPressed(PSB_PAD_LEFT)) { //如果左侧向左按钮被按下
198          LedFlip();
199          FullActRun(3, 1);
200          Timer = millis() + 50;
201          return;
202      }
203      if (ps2X.ButtonPressed(PSB_PAD_RIGHT)) { //如果左侧向右按钮被按下
204          LedFlip();
205          FullActRun(4, 1);

```

图 12.3 模式 0

```

280     if (Mode == 1)
281     {
282         if( ps2X.Button( PSB_SELECT ) )
283         {
284             if( ps2X.ButtonPressed( PSB_START ))
285             {
286                 Mode = 0;
287                 manual = TRUE;
288                 BuzzerState = 1;
289                 delay(80);
290                 manual = FALSE;
291                 delay(50);
292                 manual = TRUE;
293                 BuzzerState = 1;
294                 delay(80);
295                 manual = FALSE;
296                 LedFlip();
297                 return;
298             }
299         }
300     else
301     {
302         if (ps2X.Button(PSB_PAD_LEFT)) { //如果左侧向上按钮被按下
303             ServoPwmDutyset[6] +=30;
304             if (ServoPwmDutyset[6] > 2500)
305                 ServoPwmDutyset[6] = 2500;
306             ServoSetPluseAndTime( 6, ServoPwmDutyset[6], 60 );
307         }

```

图 12.4 模式 1

模式 1 下实现了模式切换的代码，如果不切换模式就检查 PS2 手柄被按下的按钮，然后根据按钮控制对应舵机转动。如果是 START 按钮被按下，就将所有舵机复位。