

# 幻尔科技

## ESP32 版本开发教程

### V1.0



Hiwonder 官方网站



版本号	修改日期	修改摘要
V1.0	20230923	初次发布

# 目录

1.准备工作 .....	5
1.1 接线说明 .....	5
1.2 环境配置 .....	5
2.案例开发-Python .....	5
2.1 案例 1 总线舵机信息读取 .....	5
2.1.1 运行程序 .....	6
2.1.2 实现效果 .....	7
2.1.3 案例程序简要分析 .....	8
2.2 案例 2 总线舵机 ID 设置 .....	8
2.2.1 运行程序 .....	9
2.2.2 实现效果 .....	10
2.2.3 案例程序简要分析 .....	10
2.3 案例 3 控制总线舵机转动 .....	11
2.3.1 运行程序 .....	11
2.3.2 实现效果 .....	12
2.3.3 案例程序简要分析 .....	12
2.4 案例 4 调节总线舵机速度 .....	13
2.4.1 运行程序 .....	13
2.4.2 实现效果 .....	15
2.4.3 案例程序简要分析 .....	15

2.5 案例 5 示教记录操作 .....	16
2.5.1 运行程序 .....	16
2.5.2 实现效果 .....	18
2.5.3 案例程序简要分析 .....	18
3. 案例开发-Arduino .....	20
3.1 案例 1 总线舵机信息读取 .....	20
3.1.1 运行程序 .....	20
3.1.2 实现效果 .....	20
3.1.3 案例程序简要分析 .....	21
3.2 案例 2 总线舵机 ID 设置 .....	23
3.2.1 运行程序 .....	24
3.2.2 实现效果 .....	24
3.2.3 案例程序简要分析 .....	24
3.3 案例 3 控制总线舵机转动 .....	27
3.3.1 运行程序 .....	27
3.3.2 实现效果 .....	27
3.3.3 案例程序简要分析 .....	27
3.4 案例 4 调节总线舵机速度 .....	30
3.4.1 运行程序 .....	30
3.4.2 实现效果 .....	30
3.4.3 案例程序简要分析 .....	30

3.5 案例 5 示教记录操作 .....	32
3.5.1 运行程序 .....	33
3.5.2 实现效果 .....	33
3.5.3 案例程序简要分析 .....	33

## 1. 准备工作

### 1.1 接线说明

本节示例使用的是 ESP32 单片机和 ESP32 扩展板进行开发，通过 12V 5A 适配器来供电。将总线舵机连接至 ESP32 扩展板总线舵机接口，再用数据线连接电脑和 ESP32 单片机。



### 1.2 环境配置

在电脑端安装幻尔 Python 编辑器，软件包位于“22 软件工具->ESP32 软件工具”下。关于幻尔 Python 编辑的详细使用，可在对应目录下进行学习。

在电脑端安装 Arduino IDE，软件包位于“2 软件工具->Arduino 安装包”下。关于 Arduino IDE 的详细使用，可在对应目录下进行学习。

## 2. 案例开发-Python

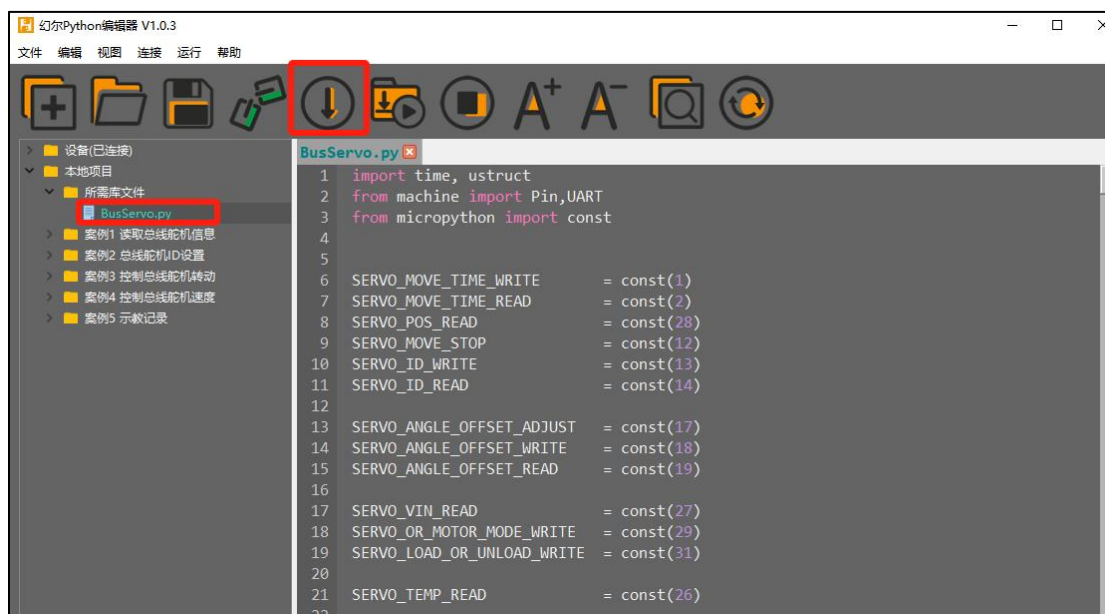
### 2.1 案例 1 总线舵机信息读取

本案例通过终端窗口显示出总线舵机 ID、位置、温度等相关信息。

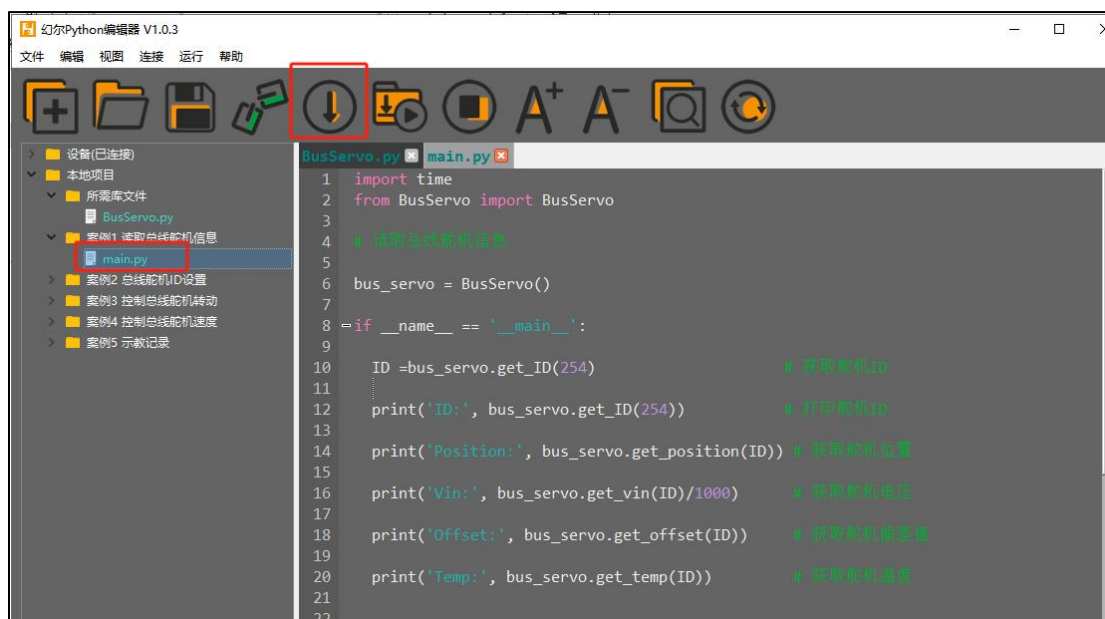
```
ID: 1
Position: 1053
Vin: 12.551
Offset: -16
Temp: 34
```

### 2.1.1 运行程序

1) 双击“BuServo.py”文件打开，点击下载按键将库文件下载到 ESP32 单片机中。



2) 双击“main.py”文件点击“下载”按键将库文件下载到 ESP32 单片机中。



3) 等待文件下载完成, 点击“设备复位”按键, 运行程序。

```
>>>
正在将文件下载到控制板.....
main.py文件下载完成!
```



### 2.1.2 实现效果

程序运行后, 终端打印舵机各项状态信息。

```
ID: 1
Position: 870
Vin: 12.531
Offset: -16
Temp: 34
```

id: 舵机 ID。在这里示例为 1。

Position: 舵机当前所在的位置。在这里示例 870 为。

dev: 舵机偏差。在这里示例为-。

Temp: 舵机当前温度。在这里示例为 34℃。

Vin: 舵机当前电压值。在这里示例为 12.531V。

### 2.1.3 案例程序简要分析

- 导入必要功能包

```
1 import time
2 from BusServo import BusServo
```

引入“time”模块，用来执行时间的操作。

引入“BusServo”功能包,主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
6 bus_servo = BusServo()
```

- 获取舵机状态信息并打印

```
8 if __name__ == '__main__':
9
10     ID = bus_servo.get_ID(254)           # 获取舵机ID
11     .....
12     print('ID:', bus_servo.get_ID(254))  # 打印舵机ID
13
14     print('Position:', bus_servo.get_position(ID)) # 获取舵机位置
15
16     print('Vin:', bus_servo.get_vin(ID)/1000) # 获取舵机电压
17
18     print('Offset:', bus_servo.get_offset(ID)) # 获取舵机偏差值
19
20     print('Temp:', bus_servo.get_temp(ID)) # 获取舵机温度
21
```

通过调用 BusServo 中的各个函数，获取舵机的各种状态信息。这些状态信息包括舵机 ID、位置、电压、偏差、当前温度。

## 2.2 案例 2 总线舵机 ID 设置

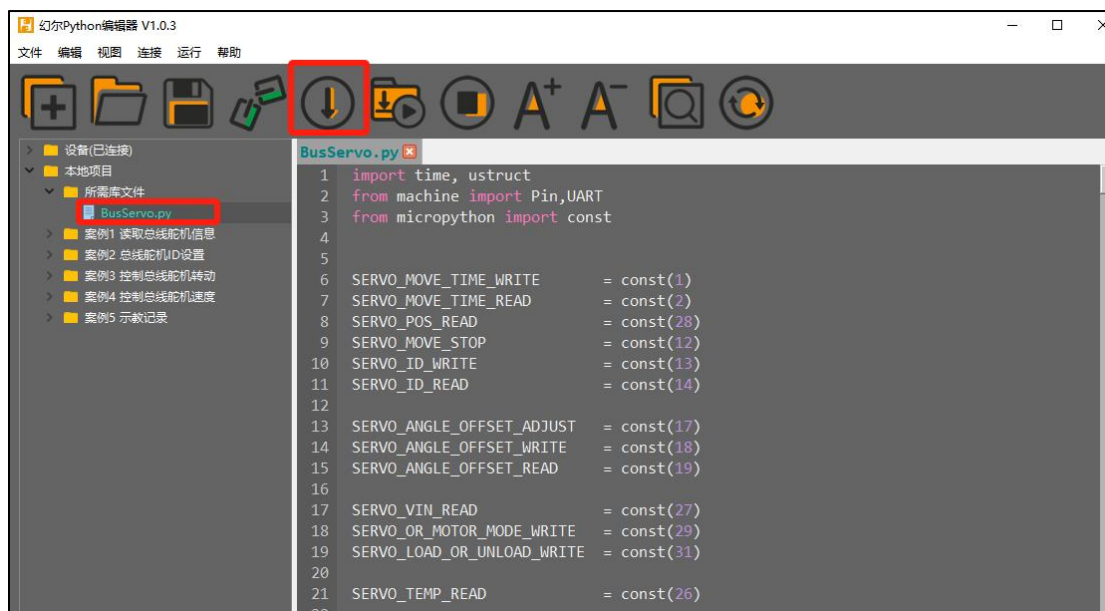
本案例修改舵机 ID 设置，并通过终端窗口显示出总线舵机新的 ID。



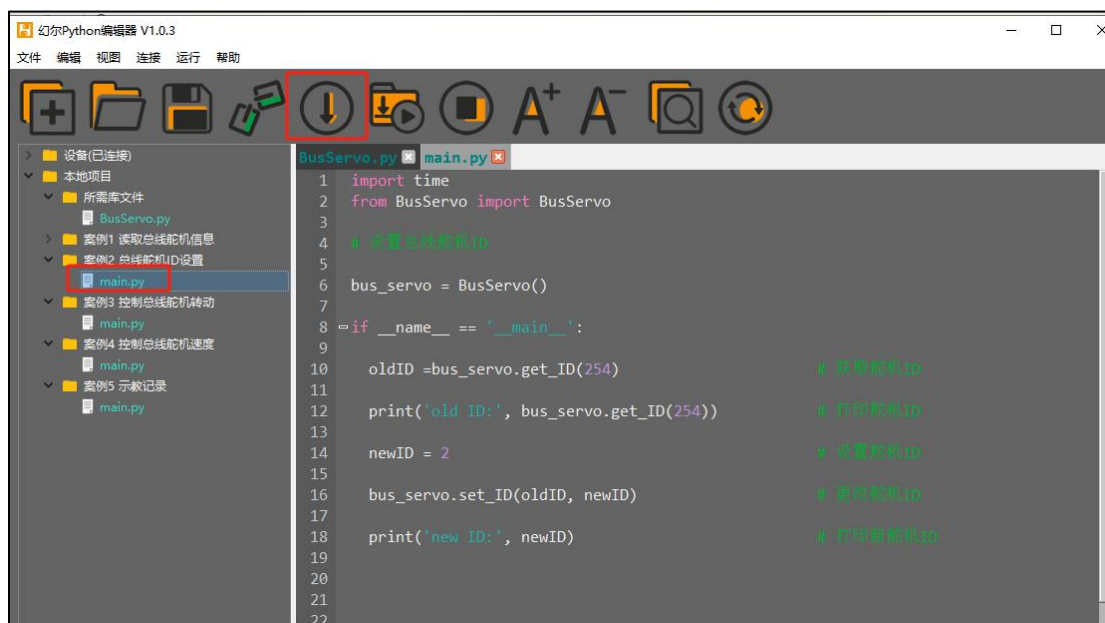
```
old ID: 1  
new ID: 2
```

## 2.2.1 运行程序

1) 双击“BusServo.py”文件打开，点击下载按键将库文件下载到 ESP32 单片机中。



3) 双击“main.py”文件点击“下载”按键将库文件下载到 ESP32 单片机中。



3) 等待文件下载完成，点击“设备复位”按键，运行程序。

```
>>>
正在将文件下载到控制板.....
main.py文件下载完成!
```



### 2.2.2 实现效果

程序运行后，并通过终端窗口显示出总线舵机旧的和新的 ID。

```
old ID: 1
new ID: 2
```

old ID:表示旧的舵机 ID。在这里示例为 1。

new ID:表示新的舵机 ID。在这里示例为 2。

### 2.2.3 案例程序简要分析

- 导入必要功能包

```
1 import time
2 from BusServo import BusServo
```

引入“time”模块，用来执行时间的操作。

引入“BusServo”功能包,主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
6 bus_servo = BusServo()
```

- 获取舵机 ID 并且打印出来

```
10 oldID =bus_servo.get_ID(254)           # 获取舵机ID
11
12 print('old ID:', bus_servo.get_ID(254)) # 打印舵机ID
```

通过调用 `bus_servo.get_ID()` 函数，将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 254，在总线舵机通信协议中表示为广播 ID，可用于对未知 ID 的舵机进行读取 ID 值。将旧的舵机 ID 值打印出来。

- 设置舵机新 ID 并且打印出来

```
14 newID = 2 # 设置舵机ID
15
16 bus_servo.set_ID(oldID, newID) # 更改舵机ID
17
18 print('new ID:', newID) # 打印新舵机ID
19
```

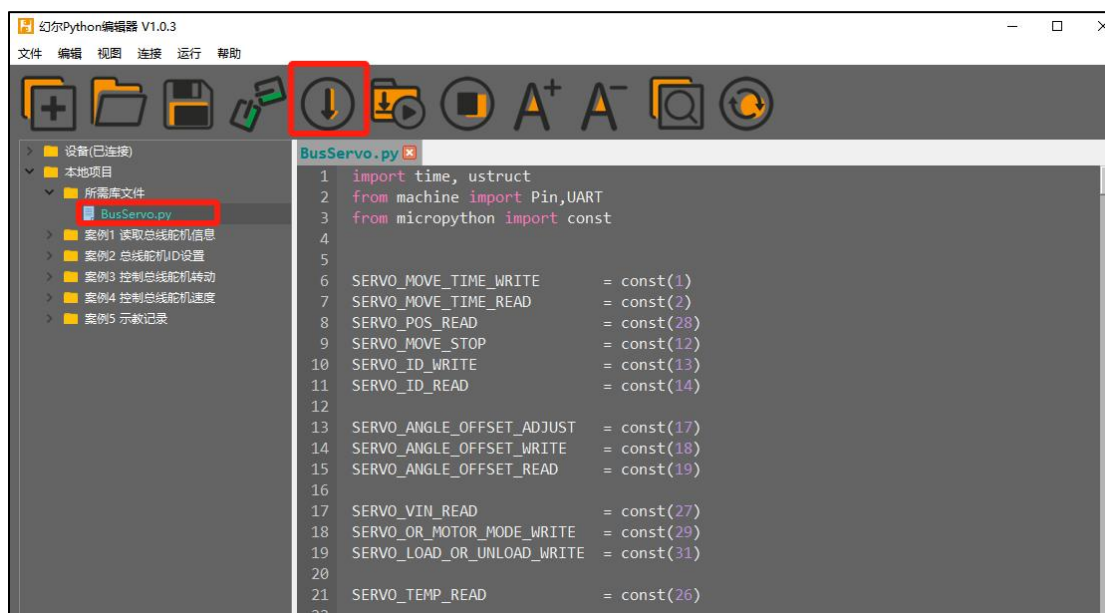
通过调用 `bus_servo.set_ID()` 函数，将连接在总线舵机调试板上的舵机的 ID 值更改为 “newID” 的值。将新的舵机 ID 值打印出来。

## 2.3 案例 3 控制总线舵机转动

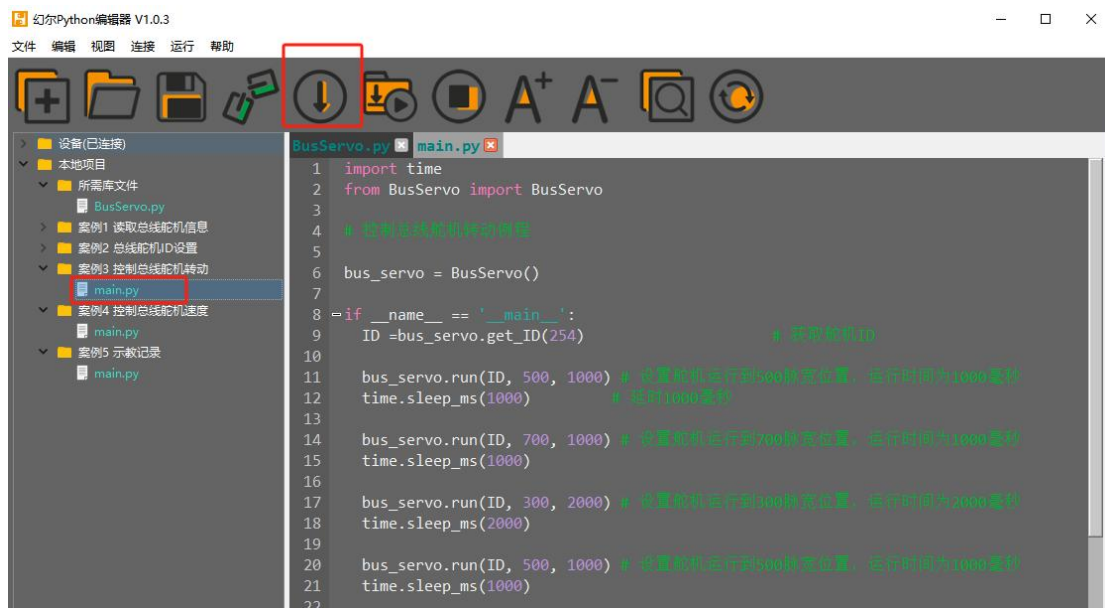
本案例 ESP32 单片机控制舵机从位置 500、1000、0、500 以 1 秒间隔转动。

### 2.3.1 运行程序

1) 双击 “BusServo.py” 文件打开，点击下载按钮将库文件下载到 ESP32 单片机中。



4) 双击 “main.py” 文件点击 “下载” 按钮将库文件下载到 ESP32 单片机中。



3) 等待文件下载完成, 点击“设备复位”按键, 运行程序。

```
>>>
正在将文件下载到控制板.....
main.py文件下载完成!
```



### 2.3.2 实现效果

程序运行后, 舵机从位置 500、1000、0、500 以 1 秒间隔转动。

### 2.3.3 案例程序简要分析

#### ● 导入必要功能包

```
1 import time
2 from BusServo import BusServo
```

引入“time”模块, 用来执行时间的操作。

引入“BusServo”功能包, 主要封装用于总线舵机通信的各功能模块, 我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
6 bus_servo = BusServo()
```

- 获取舵机 ID

```
9 ID = bus_servo.get_ID(254) # 获取舵机ID
10
```

通过调用 `bus_servo.get_ID()` 函数，将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 254，在总线舵机通信协议中表示为广播 ID，可用于对未知 ID 的舵机进行读取 ID 值。

- 控制舵机转动

```
11 bus_servo.run(ID, 500, 1000) # 设置舵机运行到500刻度位置，运行时间为1000毫秒
12 time.sleep_ms(1000) # 延时1000毫秒
13
14 bus_servo.run(ID, 1000, 1000) # 设置舵机运行到1000刻度位置，运行时间为1000毫秒
15 time.sleep_ms(1000)
16
17 bus_servo.run(ID, 0, 2000) # 设置舵机运行到0刻度位置，运行时间为2000毫秒
18 time.sleep_ms(2000)
19
20 bus_servo.run(ID, 500, 1000) # 设置舵机运行到500刻度位置，运行时间为1000毫秒
21 time.sleep_ms(1000)
```

通过调用 `bus_servo.run()` 函数，控制舵机转动。上述过程主要实现了，舵机以 1 秒的时间转动到位置 0，延时 1 秒后，再以 1 秒的时间转动到位置 1000，延时 1 秒后，再以 1 秒的时间转动到位置 0，延时 1 秒后，再以 1 秒的时间转动到位置 500。

舵机转动范围为：0-1000，对应角度为：0° - 240°。

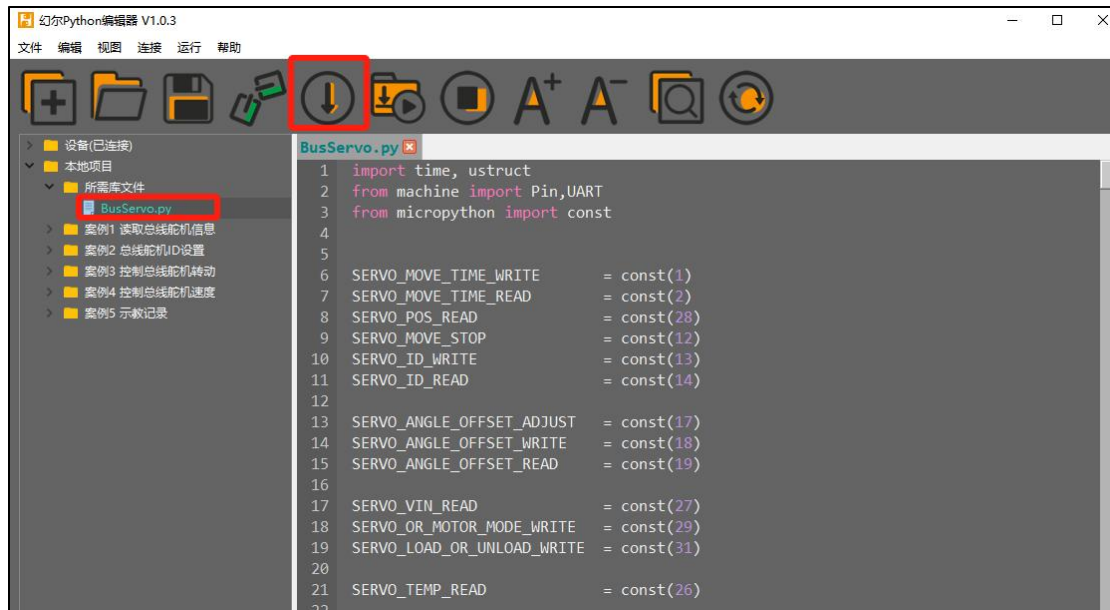
## 2.4 案例 4 调节总线舵机速度

### 2.4.1 运行程序

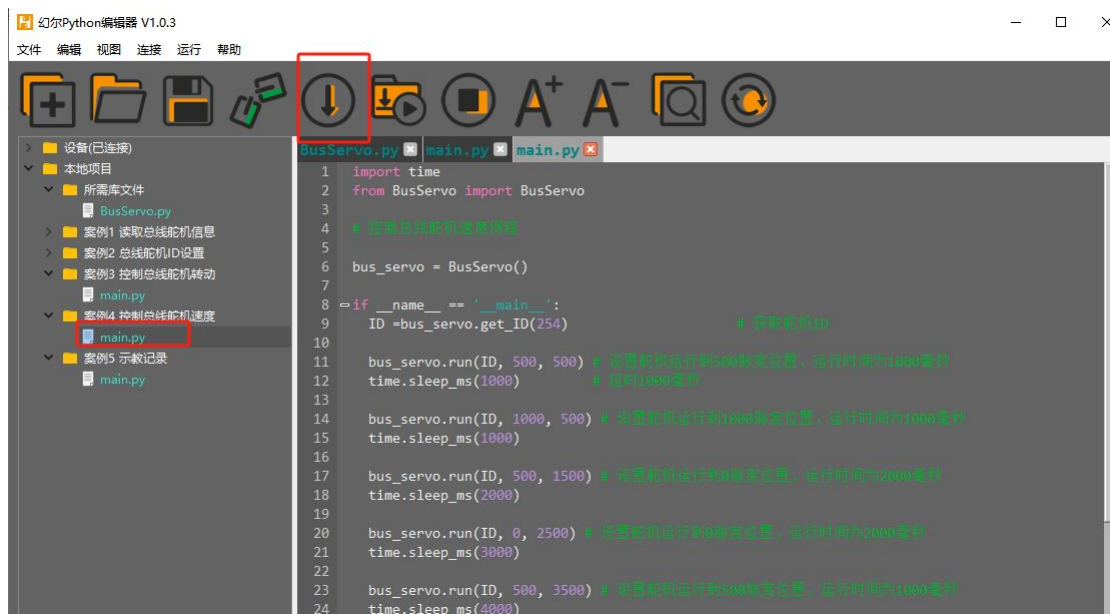
本案例通过 ESP32 单片机控制舵机以不同的速度进行转动。

双击“`BuServo.py`”文件打开，点击下载按钮将库文件下载到 ESP32 单片机中。





5) 双击“main.py”文件点击“下载”按键将库文件下载到 ESP32 单片机中。



3) 等待文件下载完成，点击“设备复位”按键，运行程序。

```
>>>
正在将文件下载到控制板.....
main.py文件下载完成!
```



### 2.4.2 实现效果

程序运行后舵机现象如下：

舵机从位置 500 开始；

以 0.5 秒的时间转动到位置 1000；

以 1.5 秒的时间转动到位置 500；

以 2.5 秒的时间转动到位置 0；

以 3.5 秒的时间转动到位置 500。

### 2.4.3 案例程序简要分析

- 导入必要功能包

```
1 import time
2 from BusServo import BusServo
```

引入“time”模块，用来执行时间的操作。

引入“BusServo”功能包，主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
6 bus_servo = BusServo()
```

- 获取舵机 ID

```
9 ID = bus_servo.get_ID(254) # 获取舵机ID
10
```

通过调用 bus\_servo.get\_ID() 函数，将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 254，在总线舵机通信协议中表示为广播 ID，可用于对未知 ID 的舵机

进行读取 ID 值。

- 控制舵机转动

```
11 bus_servo.run(ID, 500, 500) # 设置舵机运行到500脉冲位置，运行时间为500毫秒
12 time.sleep_ms(1000)        # 延时1000毫秒
13
14 bus_servo.run(ID, 1000, 500) # 设置舵机运行到1000脉冲位置，运行时间为500毫秒
15 time.sleep_ms(1000)
16
17 bus_servo.run(ID, 500, 1500) # 设置舵机运行到500脉冲位置，运行时间为1500毫秒
18 time.sleep_ms(2000)
19
20 bus_servo.run(ID, 0, 2500)   # 设置舵机运行到0脉冲位置，运行时间为2500毫秒
21 time.sleep_ms(3000)
22
23 bus_servo.run(ID, 500, 3500) # 设置舵机运行到500脉冲位置，运行时间为3500毫秒
24 time.sleep_ms(4000)
```

通过控制舵机的运行时间来控制舵机的速度。通过调用 `bus_servo.run()` 函数，控制舵机转动。上述过程主要实现了，以 0.5 秒的时间转动到位置 500，延时 1 秒后，以 0.5 秒的时间转动到位置 1000，延时 2 秒后，以 1.5 秒的时间转动到位置 500，延时 3 秒后，以 2.5 秒的时间转动到位置 0，延时 4 秒后，以 3.5 秒的时间转动到位置 500。

舵机转动范围为：0-1000，对应角度为：0° -240° 。

## 2.5 案例 5 示教记录操作

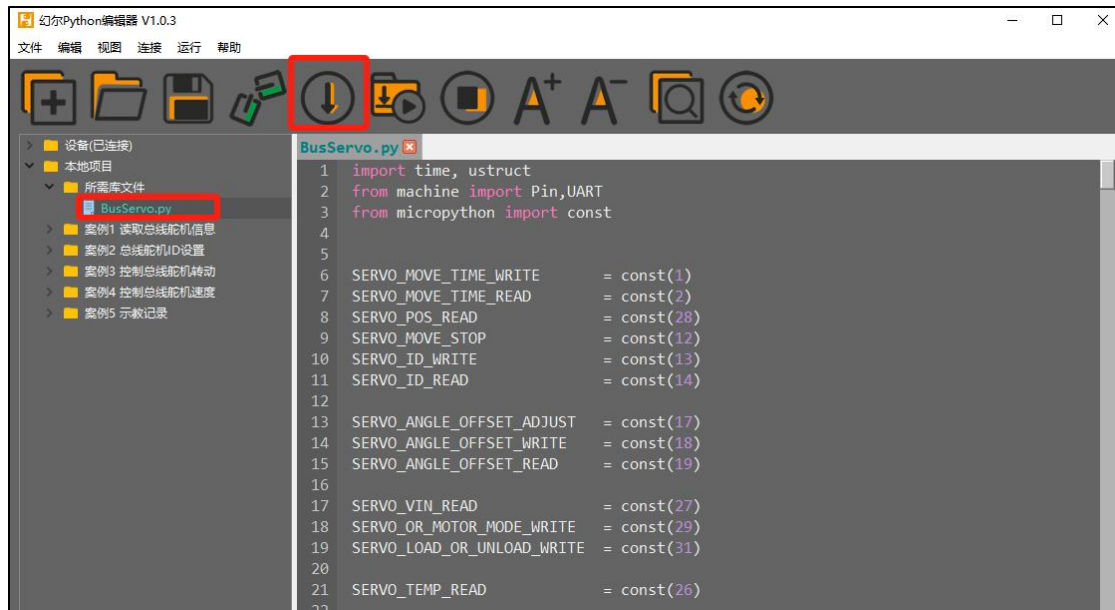
本案例 ESP32 单片机控制舵机，通过存储位置，让舵机转动到指定角度。

### 2.5.1 运行程序

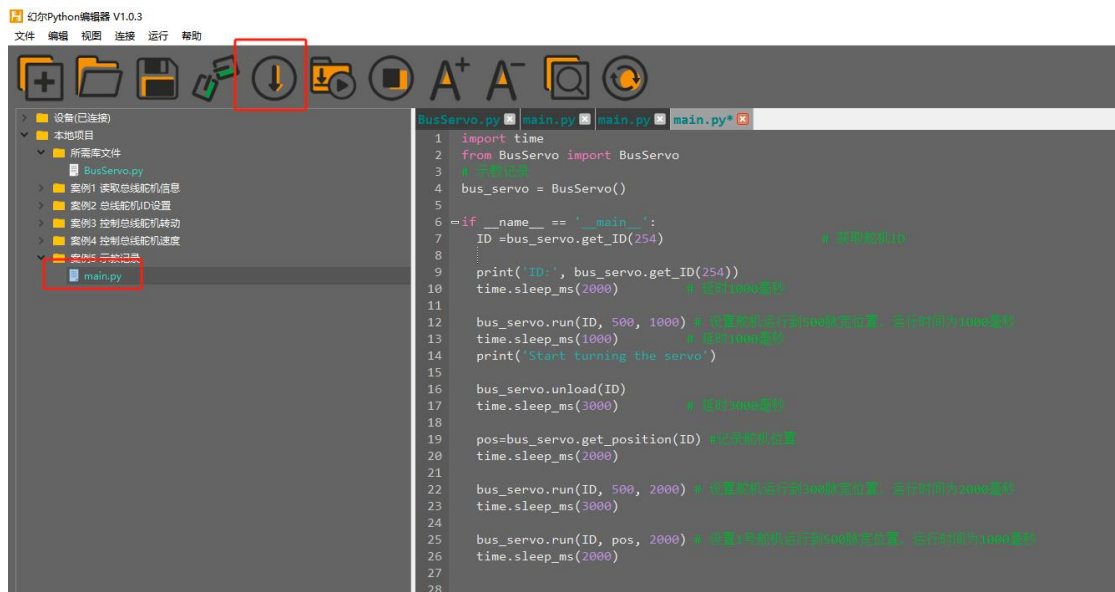
本案例通过 ESP32 单片机控制舵机以不同的速度进行转动。

- 1) 双击“BuServo.py”文件打开，点击下载按键将库文件下载到 ESP32 单片机中。





2) 双击“main.py”文件点击“下载”按钮将库文件下载到 ESP32 单片机中。



3) 等待文件下载完成，点击“设备复位”按钮，运行程序。

```
>>>
正在将文件下载到控制板.....
main.py文件下载完成!
```



### 2.5.2 实现效果

程序运行后,终端打印出舵机的 ID 编号,舵机回到 500 脉冲宽度的位置,打印出“**Start turning the servo**”信息后开始舵机开始掉电,接着我们可以用手掰动舵臂,之后舵机会记录下当前的位置,返回 500 脉冲宽度的位置之后再回到刚刚我们用手掰动的位置。

### 2.5.3 案例程序简要分析

- 导入必要功能包

```
1 import time
2 from BusServo import BusServo
```

引入“time”模块,用来执行时间的操作。

引入“BusServo”功能包,主要封装用于总线舵机通信的各功能模块,我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
6 bus_servo = BusServo()
```

- 获取舵机 ID

```
9 ID =bus_servo.get_ID(254) # 获取舵机ID
10
```

通过调用 bus\_servo.get\_ID()函数,将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 254,在总线舵机通信协议中表示为广播 ID,可用于对未知 ID 的舵机进行读取 ID 值。

- 控制舵机回中

```
12 bus_servo.run(ID, 500, 1000) # 设置舵机运行到500脉冲宽位置,运行时间为1000毫秒
13 time.sleep_ms(1000) # 延时1000毫秒
```

通过调用 bus\_servo.run()函数,控制舵机转动,回到 500 脉冲宽度的位置。使用 time.sleep\_ms()函数进行一秒的延时。

- 打印提示信息并且舵机掉电

```
14     print('Start turning the servo')
15
16     bus_servo.unload(ID)
17     time.sleep_ms(3000) # 延时3000毫秒
```

用 print 打印 “Start turning the servo” 提示信息。

通过调用 bus\_servo.unload() 函数，控制舵机掉电。使用 time.sleep\_ms() 函数进行三秒的延时，来让我们掰动舵臂。

- 记录舵机位置

```
19     pos=bus_servo.get_position(ID) #记录舵机位置
20     time.sleep_ms(2000)
```

通过调用 bus\_servo.get\_position() 函数，记录舵机位置，并且赋值给 “pos”。使用 time.sleep\_ms() 函数进行两秒的延时。

- 舵机回中

```
22     bus_servo.run(ID, 500, 2000) # 设置舵机运行到500脉冲位置，运行时间为2000毫秒
23     time.sleep_ms(3000)
```

通过调用 bus\_servo.run() 函数，控制舵机转动，回到 500 脉冲宽度的位置。使用 time.sleep\_ms() 函数进行三秒的延时。

- 舵机复位

```
25     bus_servo.run(ID, pos, 2000) # 设置1号舵机运行到500脉冲位置，运行时间为2000毫秒
26     time.sleep_ms(2000)
27
```

通过调用 bus\_servo.run() 函数，将刚刚记录的舵机位置 “pos” 传入控制舵机转动到记录的位置。使用 time.sleep\_ms() 函数进行两秒的延时。

## 3. 案例开发-Arduino


### 3.1 案例 1 总线舵机信息读取

本案例通过终端窗口显示出总线舵机 ID、位置、温度等相关信息。

```
ID: 1  
Position: 1053  
Vin: 12.551  
Offset: -16  
Temp: 34
```

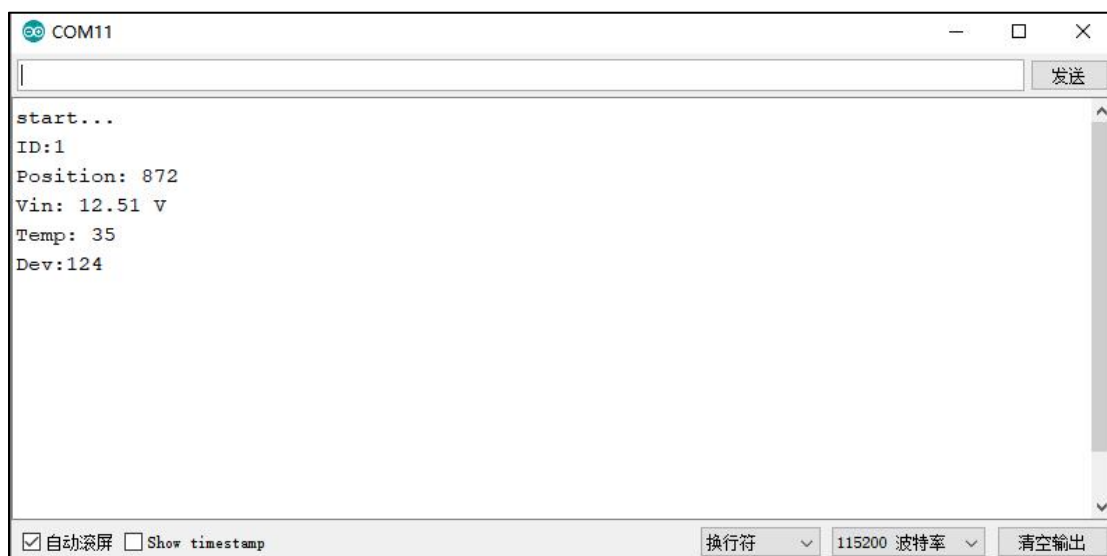
#### 3.1.1 运行程序

在“04 Arduino\案例 1 总线舵机信息读取\BusServo\_status”路径下双击打开“BusServo\_status.ino”程序。

将 ESP32 单片机连接至电脑，点击“”下载程序。（注意：如果出现上传失败的提示，可尝试将舵机调试板断开 ESP32 后上传，上传完毕后在重新接入 ESP32。）

#### 3.1.2 实现效果

程序运行后，终端打印舵机各项状态信息。



id: 舵机 ID。在这里示例为 1。

Position: 舵机当前所在的位置。在这里示例为 872。

Vin: 舵机当前电压值。在这里示例为 12.51V。

Temp: 舵机当前温度。在这里示例为 35°C。

Dev: 舵机偏差。在这里示例为 124。

### 3.1.3 案例程序简要分析

- 导入必要功能包

```
#include "LobotSerialServoControl.h" // 导入库文件
```

引入“LobotSerialServoControl.h”功能包,主要封装用于总线舵机通信的各功能模块,我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
#define SERVO_SERIAL_RX    35  
#define SERVO_SERIAL_TX    12  
#define receiveEnablePin  13  
#define transmitEnablePin 14
```

SERVO\_SERIAL\_RX: 这个宏被定义为整数 35。表示了一个接收 (RX) 数据的引脚或端口的编号。在这里,表示串行通信接口的接收引脚的引脚编号。

SERVO\_SERIAL\_TX: 这个宏被定义为整数 12。表示了串行通信接口的发送 (TX) 数据的引脚或端口的编号。

receiveEnablePin: 这个宏被定义为整数 13,表示一个接收使能引脚。在串行通信

中，使能引脚通常用于控制数据的接收操作。

transmitEnablePin: 这个宏被定义为整数 14，表示一个发送使能引脚。在串行通信中，使能引脚通常用于控制数据的发送操作。

```
HardwareSerial HardwareSerial(2);  
LobotSerialServoControl BusServo(HardwareSerial, receiveEnablePin, transmitEnablePin);
```

HardwareSerial HardwareSerial(2)创建了一个名为 HardwareSerial 的对象，并使用 2 作为参数进行初始化。表示使用第二个硬件串行（Serial）通信端口。硬件串行通信端口用于与外部设备进行串行通信。

LobotSerialServoControl BusServo 创建了一个名为 BusServo 的对象，并使用 HardwareSerial 对象、receiveEnablePin 和 transmitEnablePin 作为参数进行初始化。BusServo 对象用于控制和管理串行舵机。

### ● 初始化设置

```
void setup() {  
    // put your setup code here, to run once:  
    Serial.begin(115200);           // 设置串口波特率  
    Serial.println("start...");    // 串口打印"start..."  
    BusServo.OnInit();             // 初始化总线舵机库  
    HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX);  
    delay(500);                    // 延时500毫秒  
}
```

Serial.begin(115200): 初始化了默认串口并设置了波特率为 115200。

Serial.println("start..."): 使用默认串口打印了一条文本消息，内容为 "start..."。

BusServo.OnInit(): 调用了 BusServo 对象的 OnInit 方法。初始化总线舵机库。

HardwareSerial.begin(115200, SERIAL\_8N1, SERVO\_SERIAL\_RX, SERVO\_SERIAL\_TX): 初始化了一个名为 HardwareSerial 的串行通信对象，并设置了波特率为 115200 bps，数据位 8，无校验位，1 个停止位。



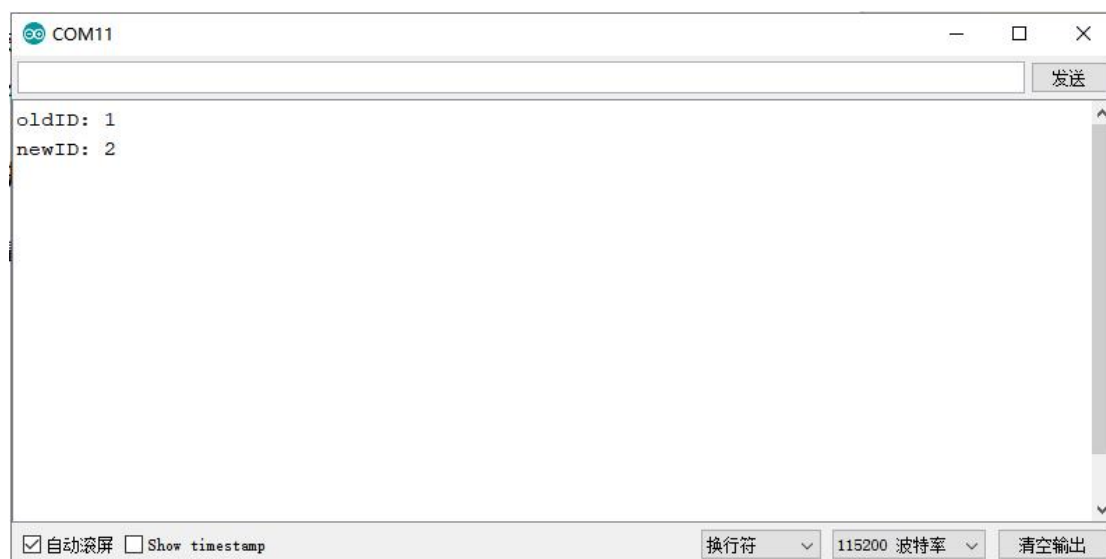
- 读取并打印相关信息

```
void loop() {  
  
    Serial.print("ID: ");  
    Serial.println(BusServo.LobotSerialServoReadID(0xFE)); // 获取舵机ID并通过串口打印  
    delay(500); // 延时  
  
    int ID = BusServo.LobotSerialServoReadID(0xFE);  
    Serial.print("Position: ");  
    Serial.println(BusServo.LobotSerialServoReadPosition(ID)); // 获取舵机位置并通过串口打印  
    delay(500); // 延时  
  
    Serial.print("Vin: ");  
    Serial.print(BusServo.LobotSerialServoReadVin(ID)/1000.0); // 获取舵机电压并通过串口打印  
    Serial.println(" V");  
    delay(900); // 延时  
  
    Serial.print("Temp: ");  
    Serial.println(BusServo.LobotSerialServoReadTemp(ID)); // 获取舵机温度并通过串口打印  
    delay(500); // 延时  
  
    Serial.print("Dev: ");  
    Serial.println(BusServo.LobotSerialServoReadDev(ID)); // 获取舵机偏差并通过串口打印  
    delay(1000); // 延时  
  
}
```

通过调用上述函数，获取舵机的各种状态信息。这些状态信息包括舵机 ID、位置、电压、偏差、当前温度，并将其打印出来。


## 3.2 案例 2 总线舵机 ID 设置

本案例修改舵机 ID 设置，并通过终端窗口显示出总线舵机新的 ID。



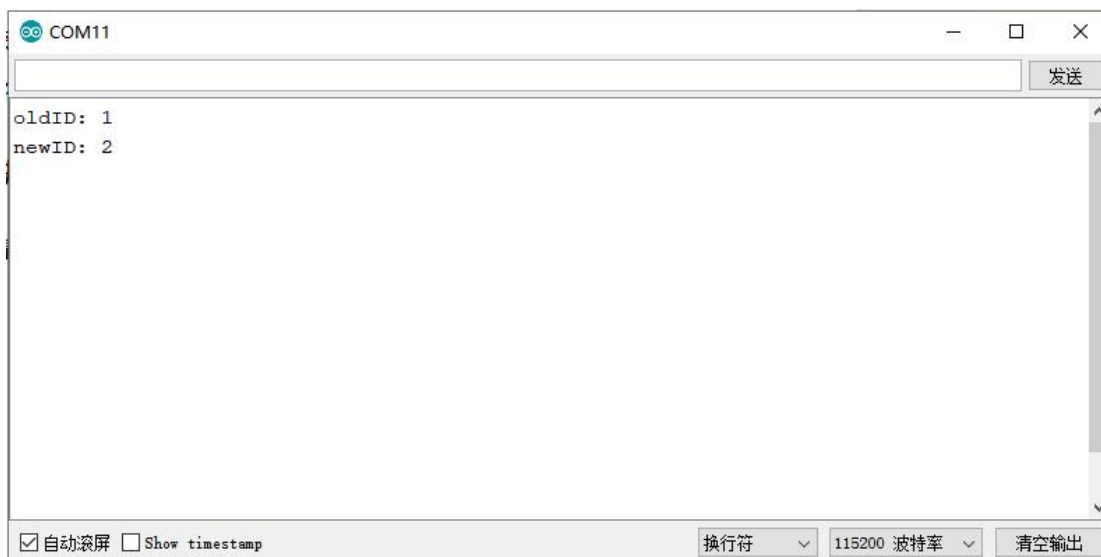
### 3.2.1 运行程序

在“04 Arduino\案例 2 总线舵机 ID 设置\BusServo\_setID”路径下双击打开“BusServo\_setID.ino”程序。

将 ESP32 单片机连接至电脑，点击“”下载程序。（注意：如果出现上传失败的提示，可尝试将舵机调试板断开 ESP32 后上传，上传完毕后在重新接入 ESP32。）

### 3.2.2 实现效果

程序运行后，并通过终端窗口显示出总线舵机旧的和新的 ID。



old ID:表示旧的舵机 ID。在这里示例为 1。

new ID:表示旧的舵机 ID。在这里示例为 1。

### 3.2.3 案例程序简要分析

- 导入必要功能包

```
#include "LobotSerialServoControl.h" // 导入库文件
```

引入“LobotSerialServoControl.h”功能包,主要封装用于总线舵机通信的各功能



模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
#define SERVO_SERIAL_RX    35
#define SERVO_SERIAL_TX    12
#define receiveEnablePin  13
#define transmitEnablePin 14
```

SERVO\_SERIAL\_RX: 这个宏被定义为整数 35。表示了一个接收（RX）数据的引脚或端口的编号。在这里，表示串行通信接口的接收引脚的引脚编号。

SERVO\_SERIAL\_TX: 这个宏被定义为整数 12。表示了串行通信接口的发送（TX）数据的引脚或端口的编号。

receiveEnablePin: 这个宏被定义为整数 13，表示一个接收使能引脚。在串行通信中，使能引脚通常用于控制数据的接收操作。

transmitEnablePin: 这个宏被定义为整数 14，表示一个发送使能引脚。在串行通信中，使能引脚通常用于控制数据的发送操作。

```
HardwareSerial HardwareSerial(2);
LobotSerialServoControl BusServo(HardwareSerial, receiveEnablePin, transmitEnablePin);
```

HardwareSerial HardwareSerial(2)创建了一个名为 HardwareSerial 的对象，并使用 2 作为参数进行初始化。表示使用第二个硬件串行（Serial）通信端口。硬件串行通信端口用于与外部设备进行串行通信。

LobotSerialServoControl BusServo 创建了一个名为 BusServo 的对象，并使用 HardwareSerial 对象、receiveEnablePin 和 transmitEnablePin 作为参数进行初始化。BusServo 对象用于控制和管理串行舵机。

- 初始化设置

```
void setup() {  
    // put your setup code here, to run once:  
    Serial.begin(115200);           // 设置串口波特率  
    Serial.println("start...");    // 串口打印"start..."  
    BusServo.OnInit();             // 初始化总线舵机库  
    HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX);  
    delay(500);                   // 延时500毫秒  
}
```

Serial.begin(115200): 初始化了默认串口并设置了波特率为 115200。

Serial.println("start..."): 使用默认串口打印了一条文本消息, 内容为 "start..."。

BusServo.OnInit(): 调用了 BusServo 对象的 OnInit 方法。初始化总线舵机库。

HardwareSerial.begin(115200, SERIAL\_8N1, SERVO\_SERIAL\_RX, SERVO\_SERIAL\_TX): 初始化了一个名为 HardwareSerial 的串行通信对象, 并设置了波特率为 115200 bps, 数据位 8, 无校验位, 1 个停止位。

- 获取舵机 ID 并且打印出来

```
Serial.print("oldID: ");  
Serial.println(BusServo.LobotSerialServoReadID(0xFE)); // 获取舵机ID并通过串口打印  
delay(1000); // 延时  
  
uint8_t oldID = BusServo.LobotSerialServoReadID(0xFE);  
delay(1000); // 延时
```

通过调用 BusServo.LobotSerialServoReadID ( ) 函数, 将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 0xFE, 换算成十进制就是 254, 在总线舵机通信协议中表示为广播 ID, 可用于对未知 ID 的舵机进行读取 ID 值。将旧的舵机 ID 值打印出来。

- 设置舵机新 ID 并且打印出来

```
uint8_t newID =2;
BusServo.LobotSerialServoSetID(oldID,newID);
delay(1000); // 延时

Serial.print("newID: ");
Serial.println(String(newID)); // 获取舵机位置并通过串口打印
delay(500); // 延时
```


通过调用 BusServo.LobotSerialServoSetID() 函数，将连接在总线舵机调试板上的舵机的 ID 值更改为 “newID” 的值。将新的舵机 ID 值打印出来。

### 3.3 案例 3 控制总线舵机转动

本案例 ESP32 单片机控制舵机从位置 500、1000、0、500 以 1 秒间隔转动。

#### 3.3.1 运行程序

在 “04 Arduino\案例 2 控制总线舵机转动\BusServo\_turn” 路径下双击打开 “BusServo\_turn.ino” 程序。

将 ESP32 单片机连接至电脑，点击 “” 下载程序。（注意：如果出现上传失败的提示，可尝试将舵机调试板断开 ESP32 后上传，上传完毕后在重新接入 ESP32。）

#### 3.3.2 实现效果

程序运行后，舵机从位置 500、1000、0、500 以 1 秒间隔转动。

#### 3.3.3 案例程序简要分析

- 导入必要功能包

```
#include "LobotSerialServoControl.h" // 导入库文件
```

引入 “LobotSerialServoControl.h” 功能包，主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
#define SERVO_SERIAL_RX    35
#define SERVO_SERIAL_TX    12
#define receiveEnablePin   13
#define transmitEnablePin  14
```

SERVO\_SERIAL\_RX: 这个宏被定义为整数 35。表示了一个接收 (RX) 数据的引脚或端口的编号。在这里，表示串行通信接口的接收引脚的引脚编号。

SERVO\_SERIAL\_TX: 这个宏被定义为整数 12。表示了串行通信接口的发送 (TX) 数据的引脚或端口的编号。

receiveEnablePin: 这个宏被定义为整数 13，表示一个接收使能引脚。在串行通信中，使能引脚通常用于控制数据的接收操作。

transmitEnablePin: 这个宏被定义为整数 14，表示一个发送使能引脚。在串行通信中，使能引脚通常用于控制数据的发送操作。

```
HardwareSerial HardwareSerial(2);
LobotSerialServoControl BusServo(HardwareSerial, receiveEnablePin, transmitEnablePin);
```

HardwareSerial HardwareSerial(2) 创建了一个名为 HardwareSerial 的对象，并使用 2 作为参数进行初始化。表示使用第二个硬件串行 (Serial) 通信端口。硬件串行通信端口用于与外部设备进行串行通信。

LobotSerialServoControl BusServo 创建了一个名为 BusServo 的对象，并使用 HardwareSerial 对象、receiveEnablePin 和 transmitEnablePin 作为参数进行初始化。BusServo 对象用于控制和管理串行舵机。

- 初始化设置

```
void setup() {  
    // put your setup code here, to run once:  
    Serial.begin(115200);           // 设置串口波特率  
    Serial.println("start...");     // 串口打印"start..."  
    BusServo.OnInit();              // 初始化总线舵机库  
    HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX);  
    delay(500);                     // 延时500毫秒  
}
```

Serial.begin(115200): 初始化了默认串口并设置了波特率为 115200。

Serial.println("start..."): 使用默认串口打印了一条文本消息，内容为 "start..."。

BusServo.OnInit(): 调用了 BusServo 对象的 OnInit 方法。初始化总线舵机库。

HardwareSerial.begin(115200, SERIAL\_8N1, SERVO\_SERIAL\_RX, SERVO\_SERIAL\_TX): 初始化了一个名为 HardwareSerial 的串行通信对象，并设置了波特率为 115200 bps，数据位 8，无校验位，1 个停止位。

## ● 控制舵机转动

```
BusServo.LobotSerialServoMove(1,500,1000); // 设置1号舵机运行到500脉宽位置，运行时间为1000毫秒  
delay(2000); // 延时2000毫秒  
  
BusServo.LobotSerialServoMove(1,1000,1000); // 设置1号舵机运行到1000脉宽位置，运行时间为1000毫秒  
delay(2000); // 延时2000毫秒  
  
BusServo.LobotSerialServoMove(1,0,1000); // 设置1号舵机运行到0脉宽位置，运行时间为1000毫秒  
delay(2000); // 延时2000毫秒  
  
BusServo.LobotSerialServoMove(1,500,1000); // 设置1号舵机运行到500脉宽位置，运行时间为1000毫秒  
delay(2000); // 延时2000毫秒
```


通过调用 bus\_servo.run() 函数，控制舵机转动。上述过程主要实现了，舵机以 1 秒的时间转动到位置 0，延时 1 秒后，再以 1 秒的时间转动到位置 1000，延时 1 秒后，再以 1 秒的时间转动到位置 0，延时 1 秒后，再以 1 秒的时间转动到位置 500。

舵机转动范围为：0-1000，对应角度为：0° -240° 。

## 3.4 案例 4 调节总线舵机速度

### 3.4.1 运行程序

在“04 Arduino\案例 4 控制总线舵机速度\BusServo\_speed”路径下双击打开“BusServo\_speed.ino”程序。

将 ESP32 单片机连接至电脑，点击“”下载程序。（注意：如果出现上传失败的提示，可尝试将舵机调试板断开 ESP32 后上传，上传完毕后在重新接入 ESP32。）

### 3.4.2 实现效果

程序运行后舵机现象如下：

- ① 舵机从位置 500 开始；
- ② 以 0.5 秒的时间转动到位置 1000；
- ③ 以 1.5 秒的时间转动到位置 500；
- ④ 以 2.5 秒的时间转动到位置 0；
- ⑤ 以 3.5 秒的时间转动到位置 500。

### 3.4.3 案例程序简要分析

- 导入必要功能包

```
#include "LobotSerialServoControl.h" // 导入库文件
```

引入“LobotSerialServoControl.h”功能包，主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数



```
#define SERVO_SERIAL_RX    35
#define SERVO_SERIAL_TX    12
#define receiveEnablePin   13
#define transmitEnablePin  14
```

SERVO\_SERIAL\_RX: 这个宏被定义为整数 35。表示了一个接收（RX）数据的引脚或端口的编号。在这里，表示串行通信接口的接收引脚的引脚编号。

SERVO\_SERIAL\_TX: 这个宏被定义为整数 12。表示了串行通信接口的发送（TX）数据的引脚或端口的编号。

receiveEnablePin: 这个宏被定义为整数 13，表示一个接收使能引脚。在串行通信中，使能引脚通常用于控制数据的接收操作。

transmitEnablePin: 这个宏被定义为整数 14，表示一个发送使能引脚。在串行通信中，使能引脚通常用于控制数据的发送操作。

```
HardwareSerial HardwareSerial(2);
LobotSerialServoControl BusServo(HardwareSerial, receiveEnablePin, transmitEnablePin);
```

HardwareSerial HardwareSerial(2)创建了一个名为 HardwareSerial 的对象，并使用 2 作为参数进行初始化。表示使用第二个硬件串行（Serial）通信端口。硬件串行通信端口用于与外部设备进行串行通信。

LobotSerialServoControl BusServo 创建了一个名为 BusServo 的对象，并使用 HardwareSerial 对象、receiveEnablePin 和 transmitEnablePin 作为参数进行初始化。BusServo 对象用于控制和管理串行舵机。

## ● 初始化设置

```
void setup() {
    // put your setup code here, to run once:
    Serial.begin(115200);           // 设置串口波特率
    Serial.println("start...");    // 串口打印"start..."
    BusServo.OnInit();             // 初始化总线舵机库
    HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX);
    delay(500);                   // 延时500毫秒
}
```

`Serial.begin(115200)`: 初始化了默认串口并设置了波特率为 115200。

`Serial.println("start...")`: 使用默认串口打印了一条文本消息, 内容为 "start..."。

`BusServo.OnInit()`: 调用了 `BusServo` 对象的 `OnInit` 方法。初始化总线舵机库。

`HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX)`: 初始化了一个名为 `HardwareSerial` 的串行通信对象, 并设置了波特率为 115200 bps, 数据位 8, 无校验位, 1 个停止位。

### ● 控制舵机转动

```
BusServo.LobotSerialServoMove(1,500,500); // 设置1号舵机运行到500脉宽位置, 运行时间为1000毫秒
delay(2000); // 延时1000毫秒

BusServo.LobotSerialServoMove(1,1000,500); // 设置1号舵机运行到700脉宽位置, 运行时间为1000毫秒
delay(1500); // 延时1000毫秒

BusServo.LobotSerialServoMove(1,500,1500); // 设置1号舵机运行到300脉宽位置, 运行时间为2000毫秒
delay(2000); // 延时2000毫秒

BusServo.LobotSerialServoMove(1,0,2500); // 设置1号舵机运行到500脉宽位置, 运行时间为1000毫秒
delay(3000); // 延时1000毫秒

BusServo.LobotSerialServoMove(1,500,3500); // 设置1号舵机运行到500脉宽位置, 运行时间为1000毫秒
delay(4000); // 延时1000毫秒
```

通过控制舵机的运行时间来控制舵机的速度。通过调用 `BusServo.LobotSerialServoMove()` 函数, 控制舵机转动。上述过程主要实现了, 以 0.5 秒的时间转动到位置 500, 延时 1 秒后, 以 0.5 秒的时间转动到位置 1000, 延时 2 秒后, 以 1.5 秒的时间转动到位置 500, 延时 3 秒后, 以 2.5 秒的时间转动到位置 0, 延时 4 秒后, 以 3.5 秒的时间转动到位置 500。

舵机转动范围为: 0-1000, 对应角度为: 0° -240° 。


## 3.5 案例 5 示教记录操作

本案例 ESP32 单片机控制舵机, 通过存储位置, 让舵机转动到指定角度。



### 3.5.1 运行程序

在“04 Arduino\案例 5 示教记录\BusServo\_record”路径下双击打开“BusServo\_record.ino”程序。

将 ESP32 单片机连接至电脑，点击“”下载程序。（注意：如果出现上传失败的提示，可尝试将舵机调试板断开 ESP32 后上传，上传完毕后在重新接入 ESP32。）

### 3.5.2 实现效果

程序运行后，终端打印出舵机的 ID 编号，舵机回到 500 脉冲宽度的位置，打印出“Start turning the servo”信息后开始舵机开始掉电，接着我们可以用手掰动舵臂，之后舵机会记录下当前的位置，返回 500 脉冲宽度的位置之后再回到刚刚我们用手掰动的位置。

### 3.5.3 案例程序简要分析

- 导入必要功能包

```
#include "LobotSerialServoControl.h" // 导入库文件
```

引入“LobotSerialServoControl.h”功能包，主要封装用于总线舵机通信的各功能模块，我们可以使用其中定义的变量和函数来控制舵机。

- 初始化 bus\_servo 函数

```
#define SERVO_SERIAL_RX    35  
#define SERVO_SERIAL_TX    12  
#define receiveEnablePin  13  
#define transmitEnablePin 14
```

SERVO\_SERIAL\_RX：这个宏被定义为整数 35。表示了一个接收（RX）数据的引脚或端口的编号。在这里，表示串行通信接口的接收引脚的引脚编号。

SERVO\_SERIAL\_TX：这个宏被定义为整数 12。表示了串行通信接口的发送（TX）数据

的引脚或端口的编号。

`receiveEnablePin`: 这个宏被定义为整数 13, 表示一个接收使能引脚。在串行通信中, 使能引脚通常用于控制数据的接收操作。

`transmitEnablePin`: 这个宏被定义为整数 14, 表示一个发送使能引脚。在串行通信中, 使能引脚通常用于控制数据的发送操作。

```
HardwareSerial HardwareSerial(2);  
LobotSerialServoControl BusServo(HardwareSerial, receiveEnablePin, transmitEnablePin);
```

`HardwareSerial HardwareSerial(2)` 创建了一个名为 `HardwareSerial` 的对象, 并使用 2 作为参数进行初始化。表示使用第二个硬件串行 (Serial) 通信端口。硬件串行通信端口用于与外部设备进行串行通信。

`LobotSerialServoControl BusServo` 创建了一个名为 `BusServo` 的对象, 并使用 `HardwareSerial` 对象、`receiveEnablePin` 和 `transmitEnablePin` 作为参数进行初始化。`BusServo` 对象用于控制和管理串行舵机。

## ● 初始化设置

```
void setup() {  
    // put your setup code here, to run once:  
    Serial.begin(115200);           // 设置串口波特率  
    Serial.println("start...");     // 串口打印"start..."  
    BusServo.OnInit();              // 初始化总线舵机库  
    HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX);  
    delay(500);                     // 延时500毫秒  
}
```

`Serial.begin(115200)`: 初始化了默认串口并设置了波特率为 115200。

`Serial.println("start...")`: 使用默认串口打印了一条文本消息, 内容为 "start..."。

`BusServo.OnInit()`: 调用了 `BusServo` 对象的 `OnInit` 方法。初始化总线舵机库。

`HardwareSerial.begin(115200, SERIAL_8N1, SERVO_SERIAL_RX, SERVO_SERIAL_TX)`: 初始化了一个名为 `HardwareSerial` 的串行通信对象, 并设置了波特率为 115200 b

ps, 数据位 8, 无校验位, 1 个停止位。

- 获取舵机 ID

```
Serial.print("ID: ");  
Serial.println(BusServo.LobotSerialServoReadID(0xFE)); // 获取舵机ID并通过串口打印  
delay(500); // 延时  
  
uint8_t ID =BusServo.LobotSerialServoReadID(0xFE);
```

通过调用 BusServo.LobotSerialServoReadID()函数, 将连接在总线舵机调试板上的舵机的 ID 值读取出来。此处参数值为 0xFE, 换算成十进制为 254, 在总线舵机通信协议中表示为广播 ID, 可用于对未知 ID 的舵机进行读取 ID 值。

- 控制舵机回中

```
BusServo.LobotSerialServoMove(ID,500,1000); // 设置舵机运行到500脉宽位置, 运行时间为1000毫秒  
delay(1000); // 延时1000毫秒
```

通过调用 BusServo.LobotSerialServoMove()函数, 控制舵机转动, 回到 500 脉冲宽度的位置。使用 time.sleep\_ms()函数进行一秒的延时。

- 打印提示信息并且舵机掉电

```
Serial.println("Start turning the servo");  
BusServo.LobotSerialServoUnload(ID); //设置舵机掉电三秒钟  
delay(3000); // 延时3000毫秒
```

用 println 打印 “Start turning the servo” 提示信息。

通过调用 BusServo.LobotSerialServoUnload() 函数, 控制舵机掉电。使用 time.sleep\_ms()函数进行三秒的延时, 来让我们掰动舵臂。

- 记录舵机位置

```
int16_t position=BusServo.LobotSerialServoReadPosition(ID); // 获取舵机现在的脉宽位置, 运行时间为2000毫秒  
delay(2000); // 延时2000毫秒
```

通过调用 BusServo.LobotSerialServoReadPosition()函数, 记录舵机位置, 并且

赋值给“position”。使用 time.sleep\_ms()函数进行两秒的延时。

- 舵机回中

```
BusServo.LobotSerialServoMove(ID, 500, 1000); // 设置舵机运行到500脉宽位置，运行时间为1000毫秒  
delay(2000); // 延时2000毫秒
```

通过调用 BusServo.LobotSerialServoMove()函数，控制舵机转动，回到 500 脉冲宽度的位置。使用 time.sleep\_ms()函数进行三秒的延时。

- 舵机复位

```
BusServo.LobotSerialServoMove(ID, position, 1000); // 设置舵机运行到上一轮的脉宽位置，运行时间为1000毫秒  
delay(1000); // 延时1000毫秒
```

通过调用 BusServo.LobotSerialServoMove() 函数，将刚刚记录的舵机位置“position”传入控制舵机转动到记录的位置。使用 time.sleep\_ms()函数进行两秒的延时。