

基于 Wokwi 的 Arduino 与外设的通信应用

5.1 串口通信

串口通信(Serial Communication)是一种串行通信方式,它通过顺序逐位传输数据,通常用于计算机和其他数字设备之间的数据交换。这种通信方式与并行通信相对,后者同时传输多位数据。

串口通信有以下特点。

- 传输模式:数据是一位接一位地顺序传输的。这意味着,与并行通信相比,串口通信的数据线更少,但传输速度较慢。
- 接口类型:常见的串口通信接口有 RS-232、RS-422、RS-485 等。其中,RS-232 是最常见的一种,经常用于连接计算机和外围设备。
- 信号电平:串口通信常使用不同的电平来表示二进制的 1 和 0。例如,在 RS-232 标准中,负电压表示 1,正电压表示 0。
- 波特率:串口通信的速度通常用波特率(Baud Rate)来表示,它指的是每秒钟可以传输的符号数量。波特率越高,数据传输速率越快。
- 应用场景:串口通信广泛应用于低速数据通信领域,如点对点通信、小型网络构建等。由于其简单和成本低廉的特点,使它在工业控制、传感器网络和初期的计算机通信中仍然非常常见。

该平台提供了一种向/从项目的 Arduino 代码发送/接收信息的方法。项目可以使用它来查看程序打印的调试消息,或发送命令控制项目的程序,这也为大家仿真程序调试提供不少便利。

Arduino UNO 和 MEGA 在硬件上都支持串行协议(USART)。串行监视器将自动连接到硬件串行端口并检测波特率,因此它是开箱即用的,不需要任何特殊配置。

其中,两者的串口发送引脚(TX)和串口接收引脚(RX)如图 5-1 所示。

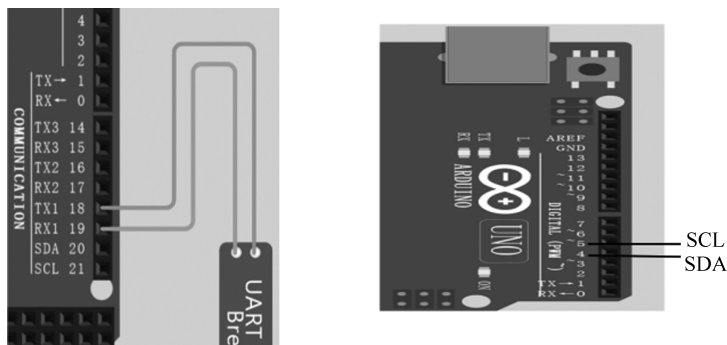


图 5-1 串口引脚

5.1.1 Arduino 与计算机通信的相关配置

```
void setup() {  
    Serial.begin(115200);           //设置相应的波特率  
    Serial.println("Hello Arduino\n");  
}  
void loop() {  
    //Do nothing...  
}
```

(1) 建立 Arduino UNO 的仿真文件,在初始化中配置好串口的波特率为 115200,并且打印“Hello Arduino”之后换行,主程序中未设置相关的操作代码,即完成 Arduino 端的配置。

(2) 而串口监视器会自动匹配相应的通信口,因为 Arduino UNO 只有一对串口通信引脚,所以,在配置 Arduino 时,无须在 diagram.json 中设置相关的引脚说明,但是对于具有多对串口通信引脚的 Arduino MEGA,则需要对其进行如下说明。

Arduino Mega 有多个硬件串行端口,项目可以通过在 diagram.json 中配置引脚,将串行监视器连接到其他串行端口。例如,要将 serial 连接到串行监视器,可在 diagram.json 中的 connections 部分添加以下行:

```
[ "mega:0", "$serialMonitor:TX", "" ],  
[ "mega:1", "$serialMonitor:RX", "" ],
```

将 MEGA 后面的引脚号改为实际 ID。

注意: 大家需将 \$serialMonitor:TX 连接到项目的串行端口的 RX 引脚,并将 \$serialMonitor:RX 连接到项目的串行端口的 TX 引脚,因为在通信过程中一方负责发送时,则另一方负责接收是一种对应的关系(交叉相连,互换角色,项目发我收,我发项目收)。

5.1.2 Arduino 串口通信函数

接下来将对 serial 类中几个用得较多的成员函数进行介绍。

`Serial.begin()`: 设置串行数据传输的数据速率(以 b/s(baud)为单位)。要与串行监视器通信时,可选其常用的波特率为 300、600、1200、2400、4800、9600、14400、19200、28800、38400、57600、115200。

可选的第二个参数用于配置数据、奇偶校验和停止位。默认值为 8 个数据位,无奇偶校验,1 个停止位,例如 `serial.begin(19200, SERIAL_5E1)` 语句设置串口波特率为 19200,数据位为 5,奇校验,停止位为 1。

`SERIAL_8N1`: (默认),其中,8 表示数据位,N 表示无校验,1 代表一个停止位。

`SERIAL_5E1`: (奇校验),其中,5 表示数据位,E 表示奇校验,1 代表一个停止位。

`SERIAL_5O1`: (偶校验),其中,5 表示数据位,O 表示偶校验,1 代表一个停止位。

`Serial.print()`: 将数据作为人们可读的 ASCII 文本打印到串行端口。此命令可以采用多种形式。数字使用每个数字的 ASCII 字符打印。浮点数同样打印为 ASCII 数字,默认为小数点后两位。字节作为单个字符发送。字符和字符串按原样发送。

`Serial.println()`: 将数据作为人类可读的 ASCII 文本打印到串行端口,后跟回车符(ASCII 13 或“\r”)和换行符(ASCII 10 或“\n”)。此命令采用与 `Serial.print()` 相同的形式,与前者区别就是该打印函数自带回车,从而方便使用者对输出数据的观察。

`Serial.write()`: 将二进制数据写入串行端口。此数据以字节或字节系列的形式发送,即显示该数据对应的 ASCII 码值对应的字符,要想直接输出表示数字的字符则应该用上述的 `print()` 之类。

将三个函数同时输出数字 65,即 65 对应 ASCII 码 A 的值,最终的输出结果如图 5-2 所示,write 输出 A,print 输出 65,println 输出 65 之后换行。

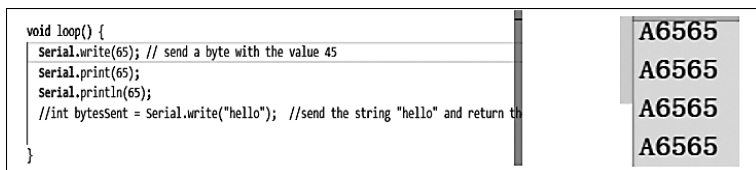


图 5-2 串口通信测试

```

Serial.read(): //读取传入的串行数据
int incomingByte = 0; //用于存储串口传输的数据
void setup() {
    Serial.begin(9600);
}
void loop() {
    if (Serial.available() > 0) {
        //读取接收数据
        incomingByte = Serial.read();
        //打印
        Serial.print("I received: ");
        //并显示对应的 ASCII 码值
        Serial.println(incomingByte, DEC);
    }
}

```

代码运行结果如图 5-3 所示。



图 5-3 代码运行结果

运行结果分析：输入 hello world, 对输入的字符转成对应的 ASCII 码值, 并使用 println 对输入的数据进行打印, 输出结果如图 5-3 所示, 可知最后两个字符为换行字符 ASCII 码值对应的内容, 符合 println 的规则。

5.1.3 串口通信案例

以下案例是基于串口通信的编码器的测试案例。

```
#define ENCODER_CLK 2 //宏定义相关引脚
#define ENCODER_DT 3
```

图 5-4 是旋转编码器 DT 和 CLK 引脚顺时针以及逆时针旋转时对应的电平信号。

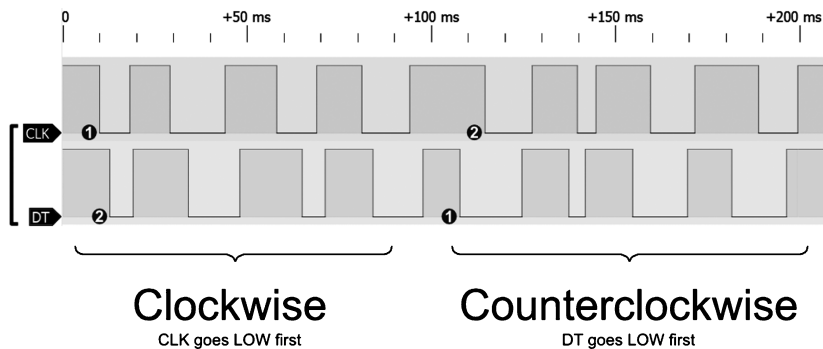


图 5-4 旋转编码器电平信号

```
void setup() {
    Serial.begin(115200); //设置串口波特率
    pinMode(ENCODER_CLK, INPUT); //编码器 A 相引脚
    pinMode(ENCODER_DT, INPUT); //编码器 B 相引脚
}

int lastClk = HIGH;

void loop() {
    int newClk = digitalRead(ENCODER_CLK); //编码脉冲技术端口
    if (newClk != lastClk) {

        lastClk = newClk;
        int dtValue = digitalRead(ENCODER_DT);
```

```

    if (newClk == LOW && dtValue == HIGH) {
        Serial.println("Rotated clockwise ▶▶");
    }
    //串口打印顺时针旋转编码信息
}
    if (newClk == LOW && dtValue == LOW) {
        Serial.println("Rotated counterclockwise ◀◀");
    }
    //打印逆时针旋转编码信息
}
}
}
}

```

运行结果如图 5-5 所示。

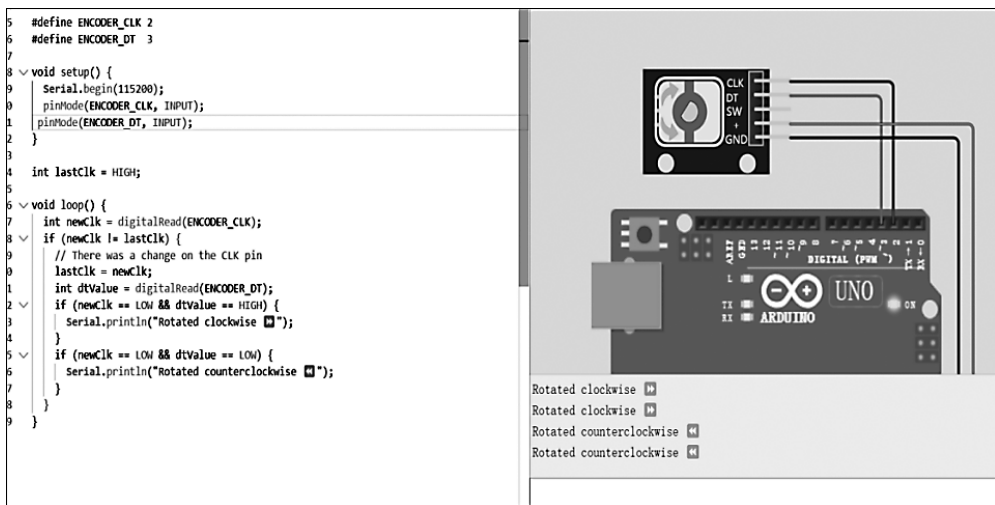


图 5-5 运行结果

5.2 I²C 总线通信

IIC 总线通信(Inter-Integrated Circuit, 通常称为 I²C, 读作“I-squared-C”)是一种串行总线,并用于连接低速度的外围设备到主板、嵌入式系统或手机。这种通信协议由飞利浦半导体(现为恩智浦半导体)在 20 世纪 80 年代初期开发。

5.2.1 I²C 总线通信的主要特点

- 总线结构: I²C 使用两条线进行通信,一条是串行数据线(SDA),用于数据传输;另一条是串行时钟线(SCL),用于同步所有设备的时钟信号。
- 多主多从结构: I²C 支持多个主设备和多个从设备。任何主设备都可以发起与任何从设备的通信。总线上的设备通过唯一的地址进行识别。
- 通信方式: 数据传输是通过 8 位字节完成的,每字节传输后都跟随一个确认位。这种通信方式使得数据传输稳定可靠。

- 速度等级：I²C 总线有不同的速度等级，包括标准模式（100kb/s）、快速模式（400kb/s）、快速模式加（1Mb/s）以及高速模式（3.4Mb/s）。
- 应用范围：I²C 广泛应用于连接低速外围设备，如传感器、存储器、显示器等，特别是在需要节省引脚数量和减少布线复杂度的嵌入式系统中。
- 简化布线：由于只需要两条线，I²C 可以大大减少设备间连接的复杂性，尤其在连接大量小型集成电路时尤为有用。

5.2.2 I²C 主机、从机和引脚

与串口通信一对一方式不同，I²C 总线通信通常有主机与从机的区别。通信时，主机负责启动和终止数据传输并且输出时钟信号；从机会被主机寻址，同时响应主机的通信请求；其中，通信速率控制由主机完成，主机通过 SCL 引脚输出时钟信号以供所有从机使用。

由主机发起所有的通信，总线设备都有对应地址；从而主机可以通过这些地址对从机的任何设备发起连接，从机响应并建立连接之后，即可进行数据传输。

Wokwi 平台中，不同的控制器 I²C 的接口位置不一致，如表 5-1 所列，数据线和时钟线的位置如图 5-6 所示。

表 5-1 I²C 接口位置

控制器型号	数据线 SDA	时钟线 SCL
UNO	A4	A5
MEGA 2560	20	21

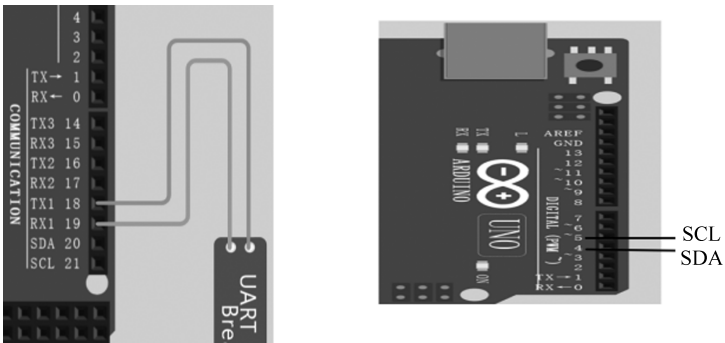


图 5-6 数据线与时钟线

5.2.3 Wire 类库成员函数

- Wire.begin()

功能：此函数初始化 Wire 库，并将 I²C 总线连接为控制器或外围设备。此函数通常只应调用一次。

参数：7 位从机地址（可选）；如果未指定，则作为控制器设备加入总线。

返回：无。

- `Wire.end()`

功能：禁用 Wire 库，反转 `Wire.begin()` 的效果。在此之后若要再次使用 Wire 库，请再次调用 `Wire.begin()`。

- `Wire.requestFrom()`

功能：控制器设备(主机)向外围设备(从机)发送数据请求信号。在此之后，从机可以使用 `onRequest()` 注册一个事件以响应主机请求；主机可以使用 `available()` 和 `read()` 函数读取字节。

语法：`Wire.requestFrom (address, quantity)` 或 `Wire.requestFrom (address, quantity, stop)`。

参数：`address` 为设备地址，`quantity` 为请求的字节数；其中，`stop` 为布尔变量，如果其为 `true`，将请求之后发送一条停止消息，释放 I²C 总线；如果其为 `false`，将在请求之后发送一条重新启动的消息，总线不会被释放，其他设备无法占用总线。

返回：无。

- `Wire.onRequest()`

功能：从机向主机注册一个事件，当从机接收到主机数据请求时触发。

语法：`Wire.onRequest(handler)`。

参数：`handler`，可被触发的事件，其中，该事件没有参数和返回值，如 `void mysensor()`。

- `Wire.onReceive()`

功能：主机向从机注册一个事件，当从机接收到主机数据请求时触发。

语法：`Wire.onReceive(handler)`。

参数：`handler`，可被触发的事件，该事件带有一个 `int` 型参数(从主机读到的字节数)且没有返回值，如 `void mysensor(int numbytes)`。

- `Wire.beginTransmission()`

功能：设定传输数据到指定地址的从机设备。之后能用 `write()` 函数发送数据，并用对应 `endTransmission()` 函数终止数据传输。

语法：`Wire.beginTransmission(address)`。

参数：`address`，要发送的从机 7 位地址。

- `Wire.endTransmission()`

功能：结束数据传输。

语法：`Wire.endTransmission()` 或 `Wire.endTransmission(stop)`。

参数：`stop` 为 `true` 或 `false`。为 `true` 将发送停止消息，在传输后释放总线；为 `false` 将发送重新启动，使连接保持活动状态。

返回：0 为成功；1 为数据太长，无法放入传输缓冲区；2 为在地址传输时收到 NACK；3 为在数据传输时收到 NACK；4 为其他错误；5 为超时。

- `Wire.write()`

功能：当为主机状态时，主机将要发送的数据加入发送队列；当为从机状态时，从机发送数据至发起请求的主机。

语法: `Wire.write(value)` `Wire.write(string)` `Wire.write(data, length)`。

参数: `value`, 要作为单字节发送的值; `string`, 要作为一系列字节发送的字符串; `data`, 要作为字节发送的数据数组; `length`, 要传输的字节数。

返回: 字节型值, 返回输入字节大小。

- `Wire.read()`

功能: 在主机, 使用 `Wire.requestFrom()` 函数发送数据请求信号后, 需要使用其获得从机字节数据; 在从机中用该函数读取主机的字节数据。

语法: `Wire.read()`。

返回: 读到的字节数据。

- `Wire.available()`

功能: 返回接收的字节数。在主机中, 通常于主机发送数据请求后使用; 在从机中, 通常于数据接收事件中使用。

语法: `Wire.available()`。

返回: 能够用于读取的字节数。

- `Wire.setWireTimeout()`

功能: 设定主机传输数据的延迟时间。

语法:

`Wire.setWireTimeout(timeout, reset_on_timeout)`

`Wire.setWireTimeout()`。

参数: `timeout` 为超时时间(以 μs 为单位), 如果为零, 则禁用超时检查。

`reset_on_timeout` 为 `true`, 则 `Wire` 硬件将在超时时自动重置; 在不带参数的情况下调用此函数时, 将配置默认超时, 该超时应足以防止在典型单主配置中锁定, 即某个操作或任务花费过长时间而导致系统无响应或锁定。

5.2.4 I²C 连接方法

I²C 的连接示意图如图 5-7 所示, 将时钟线与数据线对应连接, 并将主从设备都做共地处理(图中 GND 的地线并没有画出)。

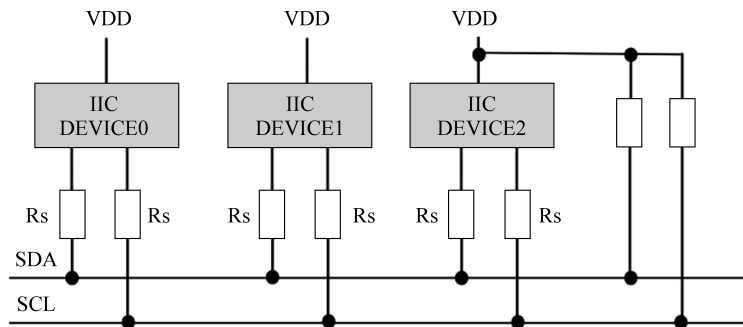


图 5-7 I²C 的连接

5.2.5 I²C 总线通信案例

如图 5-8 所示为基于陀螺仪和 OLED 模块制作的 3D 立方体模拟器,本次案例将从仿真界面开始介绍,引入仿真设计 I²C 代码介绍以及仿真测试案例。

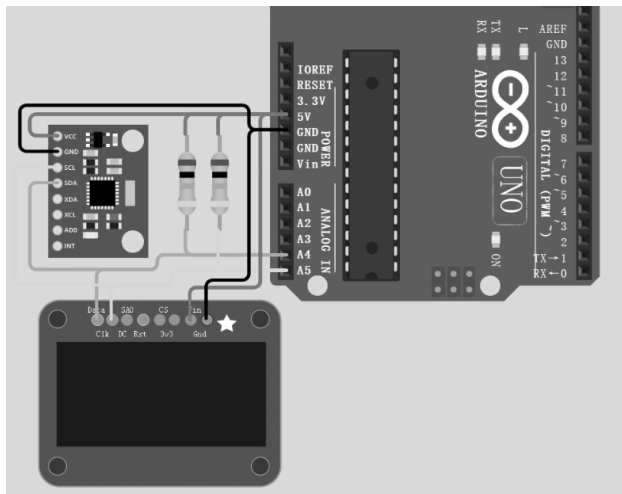


图 5-8 陀螺仪和 OLED

1. 仿真界面

仿真界面由作为 I²C 主机的 Arduino UNO 与作为从机的 OLED 和陀螺仪组成,三者共地,数据线与时钟线接上拉电阻,电源线接限流电阻。

2. 仿真设计 I²C 代码介绍

以下为 Wokwi 中涉及 I²C 的代码:

```
Wire.begin(); //未设置地址故将控制器 Arduino 设置为主机
//初始化 OLED
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C))
{
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); //程序中断
}
//初始化 mpu6050 并检测其连接
Wire.beginTransmission(mpuAddress); //开启 mpu6050 总线传输
Wire.write(0x6B); //写入 6BH 确定工作方式
Wire.write(0); //写入 0 唤醒 mpu6050
auto error = Wire.endTransmission(); //结束总线传输
Wire.beginTransmission(mpuAddress); //开启 mpu6050 总线传输
Wire.write(0x3B); //写入 3BH
Wire.endTransmission(false); //开始持续传输
Wire.requestFrom(mpuAddress, 14); //从 MPU 中读取 14 字节
```

//以下为读取陀螺仪加速度、温度以及速度信息

```
AcX=Wire.read()<<8 | Wire.read();
AcY = Wire.read()<<8 | Wire.read();
AcZ = Wire.read()<<8 | Wire.read();
Tmp=Wire.read()<<8 | Wire.read();
GyX = Wire.read()<<8 | Wire.read();
GyY = Wire.read()<<8 | Wire.read();
GyZ = Wire.read()<<8 | Wire.read();
//通过 IIC 写入清屏信息
display.clearDisplay();
//通过 IIC 写入图像信息
display.display();
```

以下为向 OLED 写指令的代码,读也同理,在 Wokwi 中以库的形式调用,故无法查看。

```
void Adafruit_SSD1306::ssd1306_command1(uint8_t c) {
    if (wire) { //IIC
        wire->beginTransaction(i2caddr); //启动 IIC 数据传输
        WIRE_WRITE((uint8_t)0x00); //总线写入数据 00H
        WIRE_WRITE(c); //总线写入数据 00H
        wire->endTransmission(); //关闭 IIC 数据传输
    } else { //选择 SPI 方式发送指令
        SSD1306_MODE_COMMAND
        SPIwrite(c);
    }
}
```

}OLED 库文件源码查看链接如下:

https://github.com/adafruit/Adafruit_SSD1306/blob/master

//以下为 OLED 头文件部分声明,在 Wokwi 中以库的形式调用,故无法查看

```
#define SSD1306_BLACK 0 //关闭线条显示
#define SSD1306_WHITE 1 //开启线条显示
//写入 8 个顶点与 12 根白色直线信息
display.drawLine(wireframe[0][0],wireframe[0][1], wireframe[1][0], wireframe
[1][1], SSD1306_WHITE);
display.drawLine(wireframe[1][0],wireframe[1][1], wireframe[2][0], wireframe
[2][1], SSD1306_WHITE);
display.drawLine(wireframe[2][0],wireframe[2][1], wireframe[3][0], wireframe
[3][1], SSD1306_WHITE);
display.drawLine(wireframe[3][0],wireframe[3][1], wireframe[0][0], wireframe
[0][1], SSD1306_WHITE);
display.drawLine(wireframe[4][0],wireframe[4][1], wireframe[5][0], wireframe
[5][1], SSD1306_WHITE);
display.drawLine(wireframe[5][0],wireframe[5][1], wireframe[6][0], wireframe
[6][1], SSD1306_WHITE);
display.drawLine(wireframe[6][0],wireframe[6][1], wireframe[7][0], wireframe
[7][1], SSD1306_WHITE);
display.drawLine(wireframe[7][0],wireframe[7][1], wireframe[4][0], wireframe
[4][1], SSD1306_WHITE);
display.drawLine(wireframe[0][0],wireframe[0][1], wireframe[4][0], wireframe
```