计算机实验教学中心

嵌入式技术课程设计实验指导手册

虚拟仿真实验篇 V1.0

（基于 Proteus 8.6 + Arduino）

哈尔滨工程大学计算机科学与技术学院

2020 年 5 月

目　录

第一篇 准备篇 …… 1
第 1 章 Arduino 简介 …… 2
1.1 Arduino 概况 …… 2
1.2 Arduino Uno 资源 …… 2
第 2 章 Proteus 入门 …… 6
2.1 Proteus 简介 …… 6
2.2 Proteus 8.6 安装 …… 6
2.3 新建 Arduino 仿真工程 …… 9
第 3 章 Arduino 仿真程序开发 …… 11
3.1 直接使用 Proteus 编写程序 …… 11
3.2 使用 Arduino IDE 生成程序文件 …… 12
3.3 使用 PlatformIO IDE 生成程序代码 …… 13
3.3.1 PlatformIO IDE 简介 …… 13
3.3.2 PlatformIO IDE 安装 …… 14
3.3.3 创建 Arduino 工程 …… 15
3.4 仿真 Arduino 程序 …… 17
第二篇 基础实验篇 …… 19
第 4 章 流水灯实验 …… 20
4.1 硬件设计 …… 20
4.2 软件设计 …… 21
4.3 仿真测试 …… 22
第 5 章 蜂鸣器实验 …… 23
5.1 硬件设计 …… 23
5.2 软件设计 …… 24
5.3 仿真测试 …… 24
第 6 章 按键输入实验 …… 25
6.1 硬件设计 …… 25
6.2 软件设计 …… 25

6.3 仿真测试 …… 26
第 7 章 虚拟串口实验 …… 27
7.1 硬件设计 …… 27
7.2 软件设计 …… 28
7.3 仿真测试 …… 29
第 8 章 外部中断实验 …… 30
8.1 硬件设计 …… 30
8.2 软件设计 …… 30
8.3 仿真测试 …… 31
第 9 章 定时器中断实验 …… 32
9.1 硬件设计 …… 32
9.2 软件设计 …… 33
9.3 仿真测试 …… 33
第 10 章 LCD1602 实验 …… 34
10.1 LCD1602 介绍 …… 34
10.2 硬件设计 …… 34
10.3 软件设计 …… 34
10.4 仿真测试 …… 35
第三篇 传感器实验篇 …… 36
第 11 章 DHT11 温湿度传感器实验 …… 37
11.1 DHT11 温湿度传感器介绍 …… 37
11.2 硬件设计 …… 37
11.3 软件设计 …… 37
11.4 仿真测试 …… 38
第 12 章 GP2D12 红外传感器实验 …… 40
12.1 GP2D12 红外传感器介绍 …… 40
12.2 硬件设计 …… 40
12.3 软件设计 …… 41
12.4 仿真测试 …… 41
第 13 章 LDR 光敏电阻实验 …… 43
13.1 硬件设计 …… 43

13.2 软件设计 …… 43
13.3 仿真测试 …… 44
第 14 章 BMP180 气压传感器实验 …… 45
14.1 BMP180 气压传感器介绍 …… 45
14.2 硬件设计 …… 45
14.3 软件设计 …… 45
14.4 仿真测试 …… 46
第四篇 FreeRTOS 实时系统篇 …… 48
第 15 章 FreeRTOS 入门 …… 49
15.1 FreeRTOS 简介 …… 49
15.2 FreeRTOS 官方文档 …… 50
15.3 FreeRTOS 源码获取 …… 50
第 16 章 FreeRTOS 任务基础知识 …… 52
16.1 裸机系统和多任务系统 …… 52
16.2 FreeRTOS 任务与协程 …… 53
16.3 FreeRTOS 任务状态 …… 54
16.4 FreeRTOS 任务优先级 …… 55
16.5 FreeRTOS 任务实现 …… 55
16.6 FreeRTOS 任务控制块 …… 56
16.7 FreeRTOS 任务堆栈 …… 57
第 17 章 任务创建与删除实验 …… 59
17.1 任务创建和删除 API 函数 …… 59
17.2 实验设计 …… 60
17.2.1 实验内容设计 …… 60
17.2.2 硬件设计 …… 60
17.2.3 软件设计 …… 60
17.3 仿真测试 …… 63
第 18 章 中断与二值信号量实验 …… 64
18.1 二值信号量简介 …… 64
18.2 创建二值信号量 …… 66
18.2.1 创建二值信号量 API 函数 …… 66

18.2.2 二值信号量创建过程分析 …… 66
18.3 释放二值信号量 …… 66
18.4 获取二值信号量 …… 67
18.5 实验设计 …… 68
18.5.1 实验内容设计 …… 68
18.5.2 硬件设计 …… 68
18.5.3 软件设计 …… 68
18.6 仿真测试 …… 70
第 19 章 优先级翻转与互斥信号量实验 …… 71
19.1 优先级翻转 …… 71
19.2 互斥信号量简介 …… 72
19.3 创建互斥信号量 …… 73
19.3.1 创建互斥信号量 API 函数 …… 73
19.3.2 互斥信号量创建过程分析 …… 73
19.4 释放互斥信号量 …… 73
19.5 获取互斥信号量 …… 74
19.6 实验设计 …… 75
19.6.1 实验内容设计 …… 75
19.6.2 硬件设计 …… 75
19.6.3 软件设计 …… 75
19.7 仿真测试 …… 78
第 20 章 消息队列实验 …… 79
20.1 消息队列简介 …… 79
20.2 创建消息队列 …… 79
20.2.1 创建队列 API 函数 …… 79
20.2.2 队列创建过程分析 …… 80
20.3 向队列发送消息（入队） …… 82
20.4 从队列读取消息（出队） …… 83
20.5 实验设计 …… 83
20.5.1 实验内容设计 …… 83
20.5.2 硬件设计 …… 84

20.5.3 软件设计 ………………………………………………………………………… 84
20.6 仿真测试 ………………………………………………………………………………… 87
第五篇 综合实验篇 ……………………………………………………………………… 88
第 21 章 智能家居照明控制系统实验 ………………………………………………… 89

第一篇 准备篇

磨刀不误砍柴工，学习嵌入式技术开发，实验平台必不可少！本篇将介绍本书的实验环境，包括硬件实验环境 Arduino Uno 平台，及虚拟仿真实验环境 Proteus 软件，并详细介绍利用 Proteus 创建 Arduino 虚拟开发环境及程序仿真方式。

本篇分为如下 3 个章节：

第 1 章 Arduino 简介

第 2 章 Proteus 入门

第 3 章 Arduino 仿真程序开发

第1章　Arduino 简介

1.1　Arduino 概况

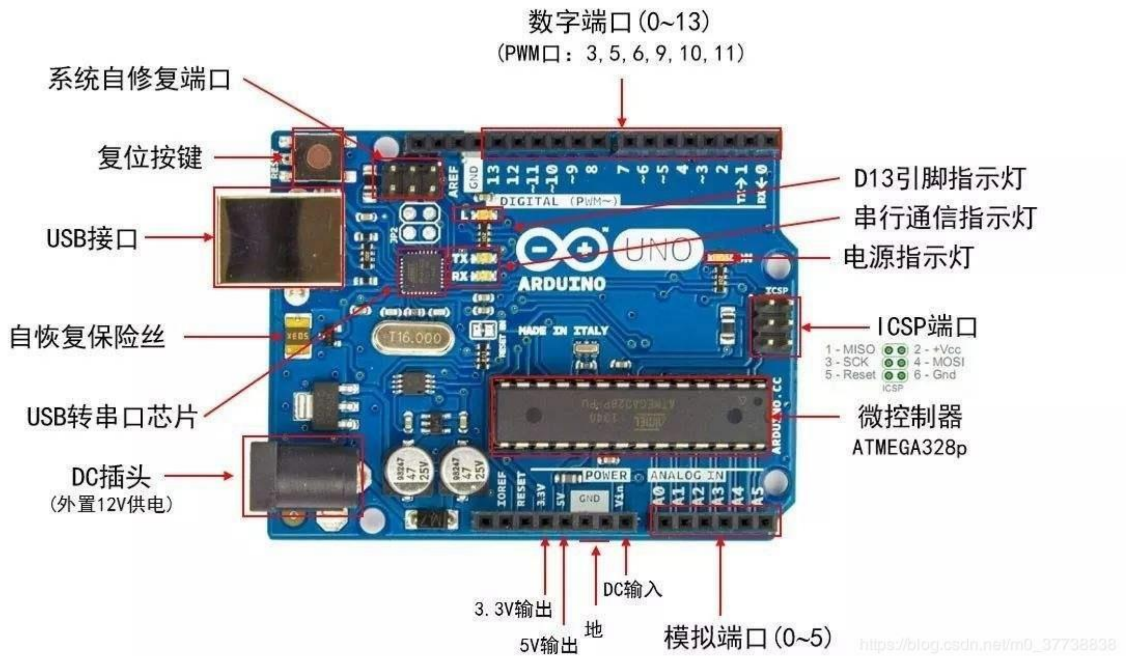
Arduino 是一个开发各类设备，感知和控制物理世界的生态系统。包括 Arduino 硬件，Arduino IDE 软件开发环境和 Arduino 社区。

Arduino 简化了使用单片机工作的流程，照比其他单片机平台，Arduino 具有以下特点：

a) 便宜：相比其他单片机平台而言，Arduino 开发板价格相对便宜，产品丰富，涵盖了从低端到高端的多种开发板。
b) 跨平台运行：Arduino IDE 能在 Windows、macOS 和 Linux 系统中运行，而大多数其他单片机系统仅限于在 Windows 操作系统中运行。
c) 简单明了的编程环境：Arduino 的编程语言和 Arduino IDE 开发环境易于初学者使用，同时对高级用户来讲也足够灵活。
d) 开源和可扩展软件：Arduino IDE 作为开源工具发布，允许有经验的程序员在其基础上进行扩展开发。所使用的编程语言可以通过 C++ 库进行扩展。同样，Arduino IDE 也支持根据需要直接将所用单片机芯片的开发代码（如 AVR-C）添加到 Arduino 程序中。
e) 开源和可扩展硬件
   Arduino 以 Atmel 公司的 ATMEGA 8 位系列单片机及其 SAM3X8E 和 SAMD21 32 位单片机为硬件基础。开发板和模块计划在遵循“知识共享许可协议”的前提下发布，所以经验丰富的电路设计人员可以做出属于自己的模块，并进行相应的扩展和改进。即使是经验相对缺乏的用户也可以做出试验版的基本 Uno 开发板，便于了解其运行的原理并节约成本。

1.2　Arduino Uno 资源

Arduino Uno 是基于 ATmega328P 的单片机开发板。该开发板由 14 路数字输入/输出引脚（其中 6 路可以用作 PWM 输出）、6 路模拟输入、1 个 16MHz 的石英晶体振荡器、一个 USB 接口、1 个电源接头、1 个 ICSP 数据头以及 1 个复位按钮组成。Uno 包含了单片机运行所需的所有要素，只需用 USB 连接线将其连接到计算机，或利用 AC-DC 适配器或电池供电后即可启动。Uno 的特色在于将 ATmega16U2 编程为一个 USB-to-serial 转换器，以便能简单、轻松和自由地安装驱动程序。



1、技术规格：

微处理器	ATmega328P
工作电压	5V
输入电压（推荐)	7-12V
输入电压（限值)	6-20V
数字输入/输出引脚	14 路（其中 6 路可用于 PWM 输出）
PWM 数字 I/ O 引脚	6
模拟输入引脚	6
每路输入/输出引脚的直流电流	20 mA
3.3V 引脚的直流电流	50 mA
闪存存储器	32KB，其中 bootloader 占用 0.5KB
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
时钟频率	16 MHz

2、电源引脚

Vin：电源输入引脚。

5V：输出 5V 电压。

3.3V：输出 3.3V 典雅，最大输出能力为 50 mA。

GND：接地引脚。

IOREF：I/O 参考电压。

3、存储空间

ATmega328 有 32KB Flash 存储空间（其中 0.5KB 用于 bootloader），2KB 的 SRAM 和 1KB 的 EEPROM。可以使用官方提供的 EEPROM 库读写 EEPROM 空间。

4、输入输出

Arduino Uno 有 14 个数字输入输出引脚，其中一些带有特殊功能，这些引脚如下：

（1）Serial：0（RX）、1（TX），被用于接收和发送串口数据。这两个引脚通过连接到 ATmega16u2 来与计算机进行串口通信。

（2）外部中断：2、3，可以输入外部中断信号。中断有四种触发模式：低电平触发、电平改变触发、上升沿触发、下降沿触发。

（3）PWM 输出：3、5、6、9、10、11，可用于输出 8-bit PWM 波。

（4）SPI：10（SS）、11（MOSI）、12（MISO）、13（SCK），可用于 SPI 通信。可以使用官方提供的 SPI 库操纵。

（5）L-LED：13。13 号引脚连接了一个 LED，当引脚输出高电平时打开 LED，当引脚输出低电平时关闭 LED。

（6）IIC：A4（SDA）、A5（SCL），可用于 IIC 通信，可以使用官方提供的 Wire 库操纵。

Arduino Uno 还有 6 个模拟输入引脚，每个模拟输入都有 10 位分辨率（即 1024 个不同的值）。默认情况下，模拟输入电压范围为 0～5V，可使用 AREF 引脚和 analogReference()函数设置其他参考电压。

此外，Arduino 还拥有复位端口。接低电平会使 Arduino 复位，同时复位按键按下时也会使该端口接到低电平，从而让 Arduino 复位。

5、指示灯（LED）

Arduino UNO 带有 4 个 LED 指示灯，作用分别如下：

ON：电源指示灯。当 Arduino 通电时，ON 灯会点亮。

TX：串口发送指示灯。当使用 USB 连接到计算机且 Arduino 向计算机传输数据时，TX 灯会点亮。

RX：串口接收指示灯。当使用 USB 连接到计算机且 Arduino 接收到计算机传来的数据时，RX 灯会点亮。

L：可编程控制指示灯。该 LED 通过特殊电路连接到 Arduino 的 13 号引脚，当 13 号引脚为高电平或高阻态时，该 LED 会点亮；低电平时，不会点亮。可以通过程序或者外部输入信号，控制该 LED 亮灭。

6、通信

Arduino UNO 具备多种通信接口，可以和计算机、其他 Arduino 或者其他控制器通信。

ATmega328 提供了 UART TTL(5V)串口通信，其位于 0(RX)和 1(TX)两个引脚上。Uno 上的 ATmega16U2 会在计算机上模拟出一个 USB 串口，使得 ATmega328 能和计算机通信。Arduino IDE 提供了串口监视器，使用它可以收发简单文本数据。Uno 上的 RX\TX 两个 LED 可以指示当前 Uno 的通信状态。

SoftwareSerial 库可以将 Uno 的任意数字引脚模拟成串口，从而进行串口通信。

ATmega328 也支持 I2C 和 SPI 通信。Arduino IDE 自带的 Wire 库，可用于驱动 I2C 总线，自带的 SPI 库，可用于 SPI 通信。

7、自动复位

一些开发板在上传程序前需要手动复位，而 Arduino Uno 的设计不需要如此，在 Arduino Uno 连接电脑后可以由程序控制其复位。在 Arduino IDE 中点击上传程序，在上传前即会触发复位，从而运行引导程序，完成程序上传。

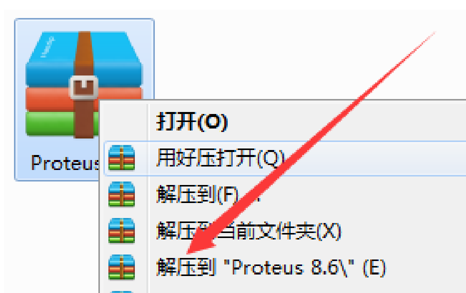
第2章 Proteus 入门

2.1 Proteus 简介

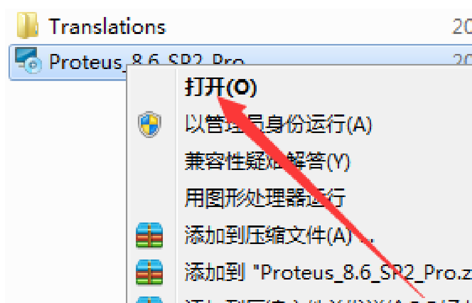
Proteus 软件是英国 Lab Center Electronics 公司出版的 EDA 工具软件。Proteus 将电路仿真软件、PCB 设计软件和虚拟模型仿真软件结合成专业的电子设计平台，主要用于各种电器、电子原件的设计与开发。该软件能支持 8051、HC11、PIC10/12/16/18/24/30/DsPIC33、AVR、ARM Cortex-M 处理器的使用，在编译方面能支持 AR、Keil 和 MATLAB 等多种编译器。

2.2 Proteus 8.6 安装

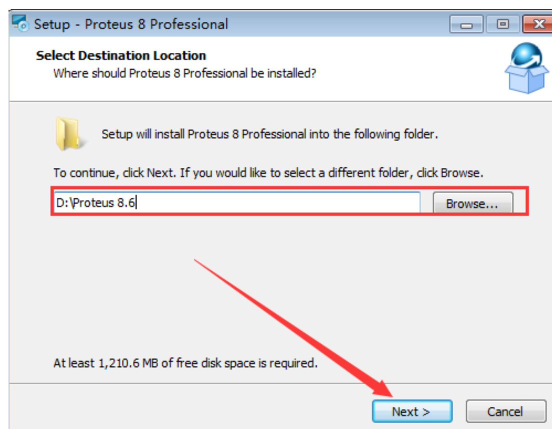
1、解压安装包 Proteus 8.6。



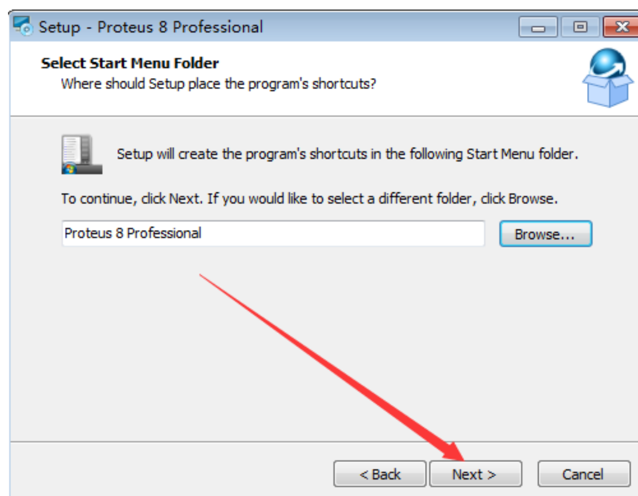
2、在解压文件夹中找到 Proteus_8.6_SP2_Pro，并打开。



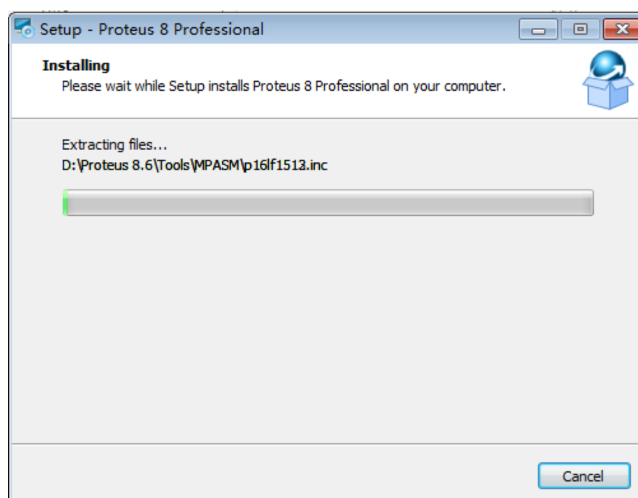
3、点击 Browse…更改安装路径，点击 Next。



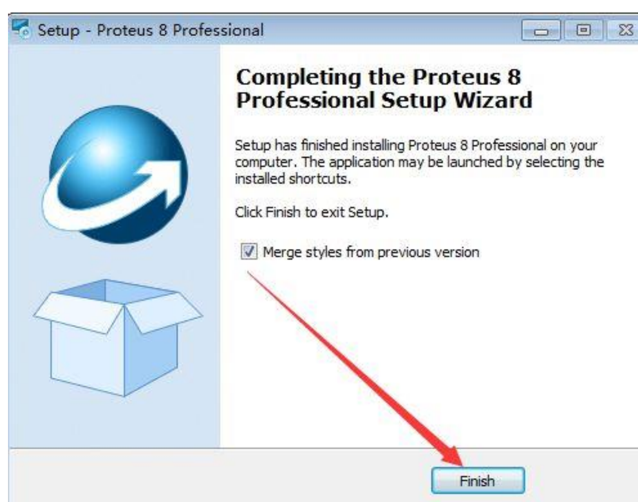
4、点击 Next。



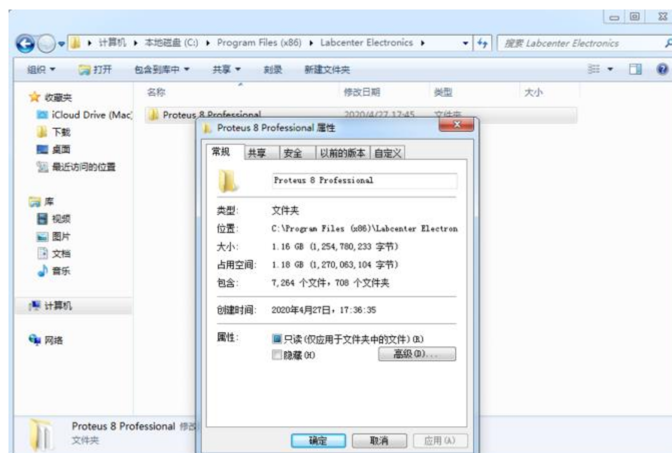
5、安装中。



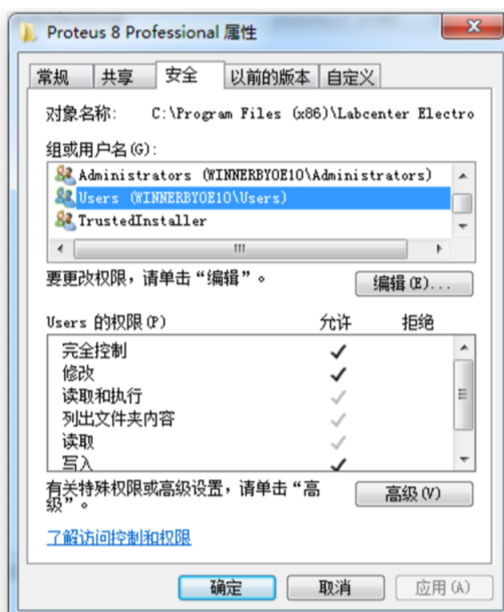
6、点击 Finish。



7、找到 Proteus 8.6 的安装文件夹，右键，选择属性。



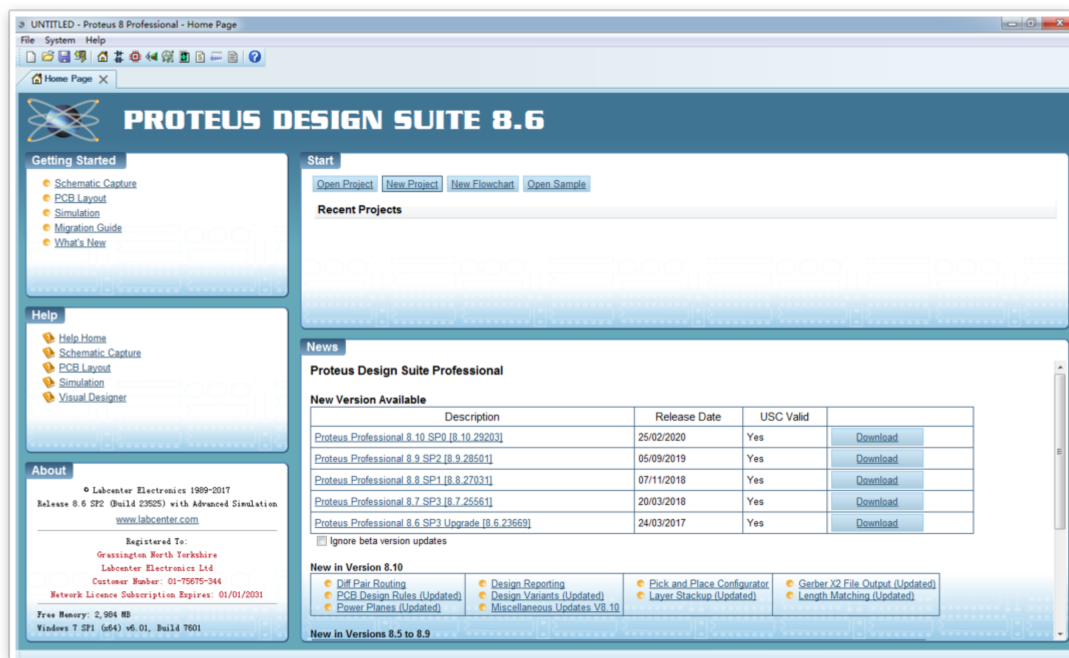
8、在“安全”选项卡中，点击编辑。



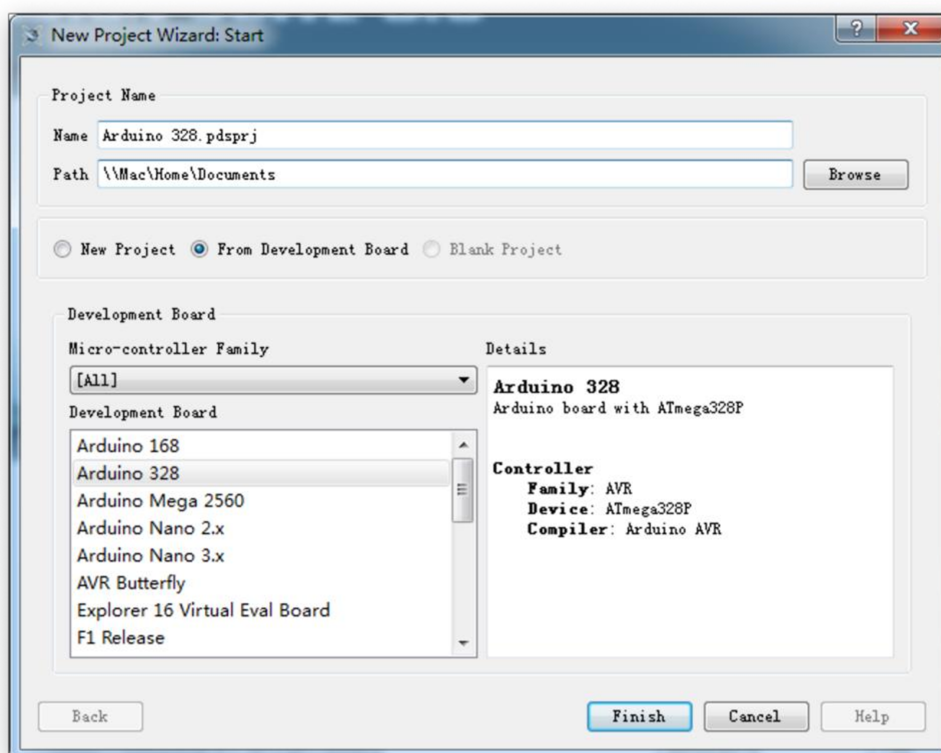
9、选择 Users，勾选“完全控制”，点击应用。

2.3 新建 Arduino 仿真工程

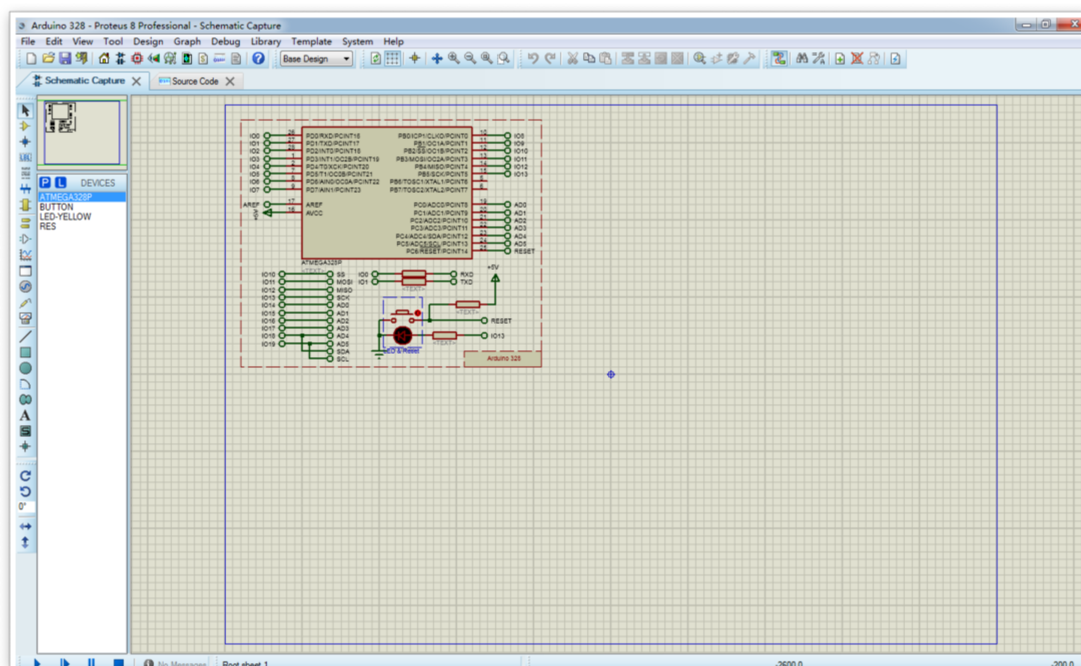
1、打开 Proteus 8 Professional，点击 Start 中的 New Project。



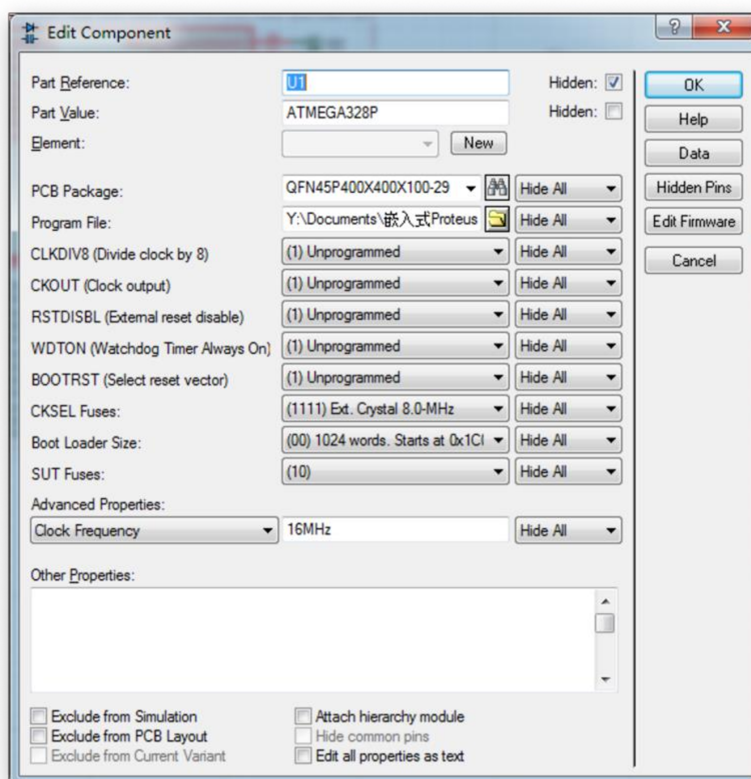
2、选择“From Development Board”，并在 Develop Board 中选择 Arduino 328，设定工程的名称和路径，点击 Finish。



3、创建完成的 Proteus Arduino Uno 仿真工程如下。



4、双击 ATMEGA328P 芯片或单击右键选择“Edit Component”，打开 Edit Component 选项卡。设置 CKSEL Fuses（熔丝位）为 1111，Clock Frequency（时钟频率）为 16MHz。另外 Program File 即为加载的 Arduino 程序文件的路径，默认是由 Proteus 直接编译 Arduino 代码所生成程序文件的路径。

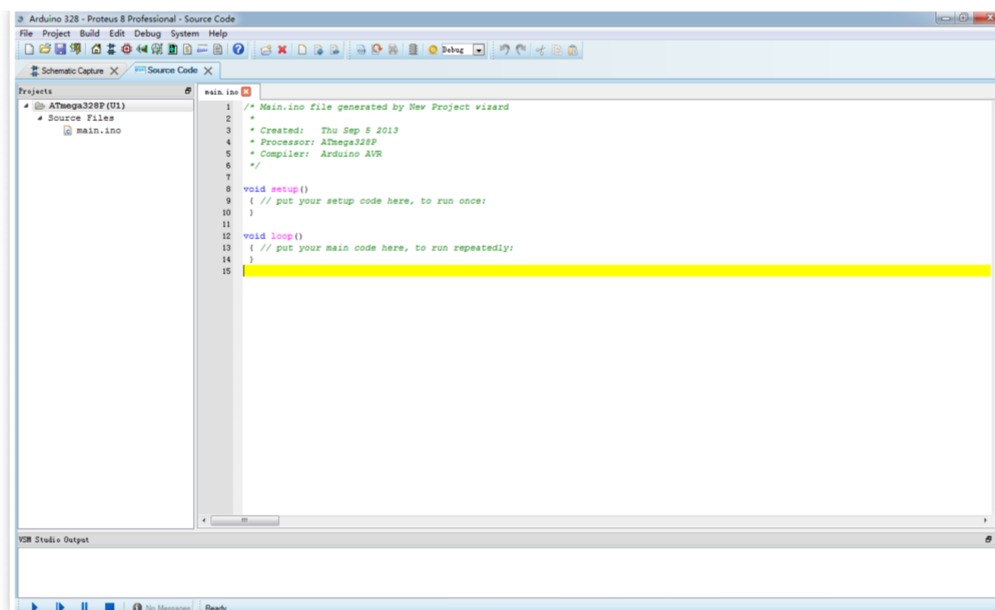


第3章　Arduino 仿真程序开发

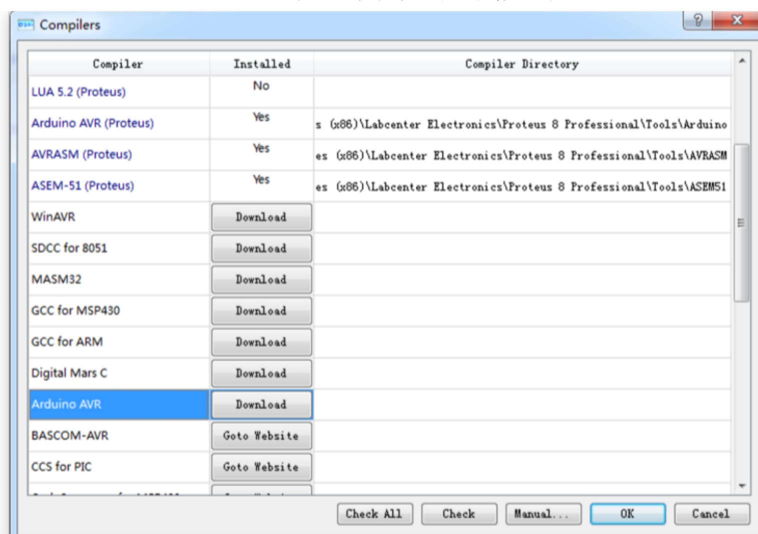
开发 Arduino 仿真程序主要有三种方法，包括使用 Proteus 软件直接编写程序、使用 Arduino IDE、使用第三方 IDE。

3.1　直接使用 Proteus 编写程序

Proteus 8.6 内置 VSM Studio，可以直接编写程序并编译生成程序文件。右键点击 ATMEGA328P 芯片选择“Edit Source Code”，或单击工具栏中的 Source Code 即可打开 Source Code 选项卡。



Proteus 编译 Arduino 工程需要安装 Arduino AVR 编译工具。点击菜单栏中的“System”，选择“Compilers Configuration”，找到“Arduino AVR”（不带 Proteus），点击“Download”即可自动下载安装。



安装完毕后，编写 Arduino 程序代码，点击工具栏中的“Build Project”即可编译生成程序文件，并默认加载到 ATMEGA328P 芯片中。

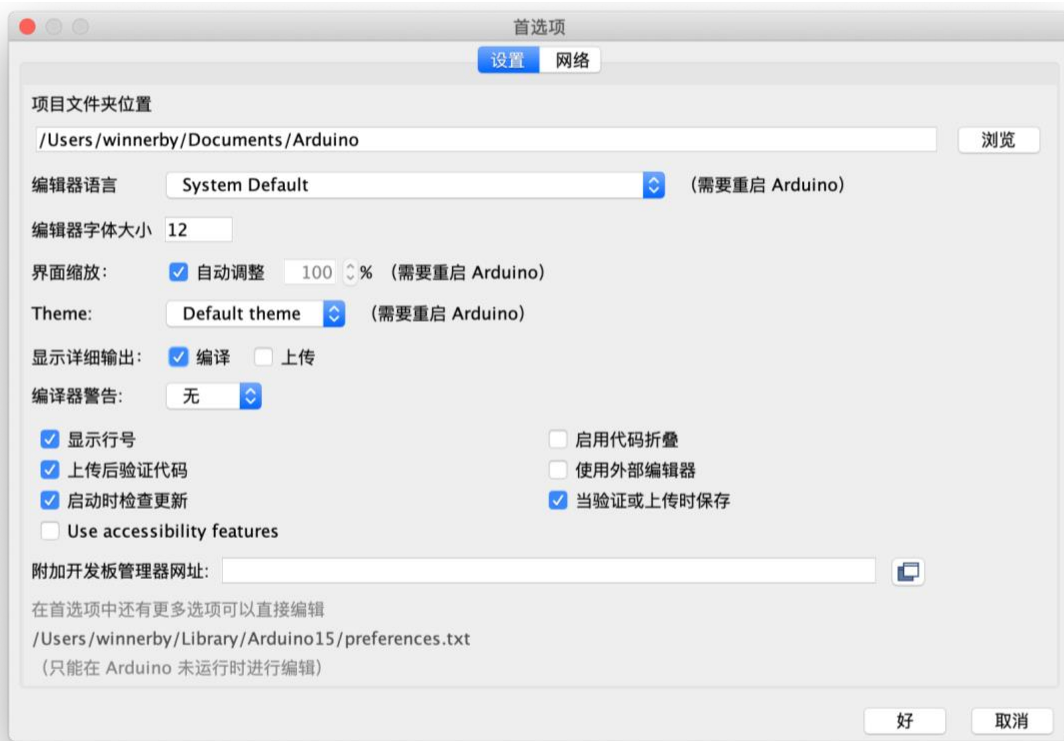
由于下载 Arduino AVR 编译工具受网络状态影响较大，不推荐采用此方法进行 Arduino 仿真程序开发。

3.2 使用 Arduino IDE 生成程序文件

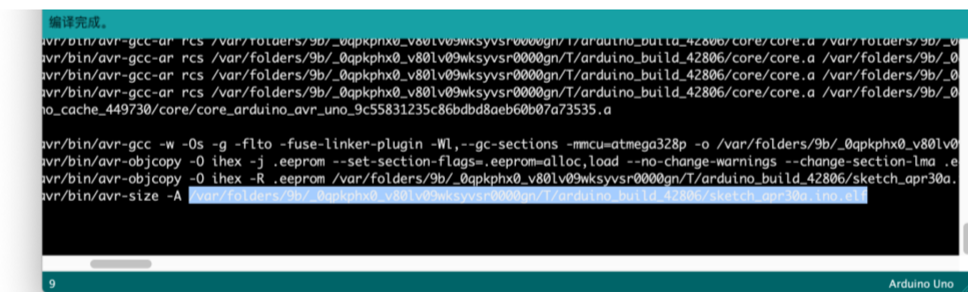
Arduino IDE 是 Arduino 官方出品的 Arduino 开发环境，支持 Windows、macOS 和 Linux 操作系统。Arduino IDE 可以从 Arduino 网站获取，也可以通过 Windows 10 应用商店获取。

使用 Arduino IDE 编写 Arduino 仿真程序需要得知生成程序文件的路径，有两种方式。

第一种方式是打开 Arduino 的首选项设置，勾选显示详细输出中，编译前面的复选框。



勾选后，在编译程序时即可输出程序文件路径。

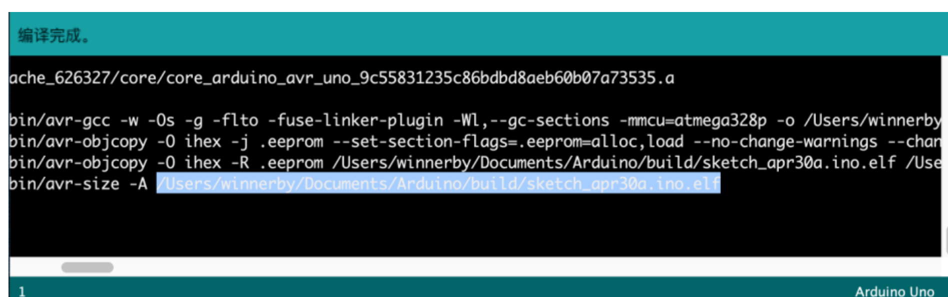


可以看出 Arduino IDE 在一个临时生成的文件夹中生成了 ELF 格式的程序文件，然而这种方式并不利于找到 ELF 文件。

另一种方式是配置 Arduino IDE 的 preferences.txt 文件，该配置文件路径在首选项窗口的最下方。在 preferences.txt 中添加一行代码：

build.path=输出 ELF 文件的路径

此时再进行编译可以看到 ELF 文件已经输出到了设置的文件夹。



3.3 使用 PlatformIO IDE 生成程序代码

3.3.1 PlatformIO IDE 简介

嵌入式与物联网开发，最不舒服的体验是，不同厂家的嵌入式芯片需要使用不同的集成开发环境，甚至个别芯片只能利用交叉编译工具链进行手动编译，另外，集成开发环境对于操作系统还有一定的限制。

PlatformIO IDE 是新一代的跨平台物联网集成开发环境，支持 Windows、macOS 和 Linux 操作系统，甚至可以在云端进行开发。PlatformIO IDE 支持 800 余种嵌入式开发板、35 余种嵌入式开发平台及 20 余种嵌入式框架。同时，它可以作为插件集成到 Visual Studio Code 或 CLion 等现有的代码编辑器中。

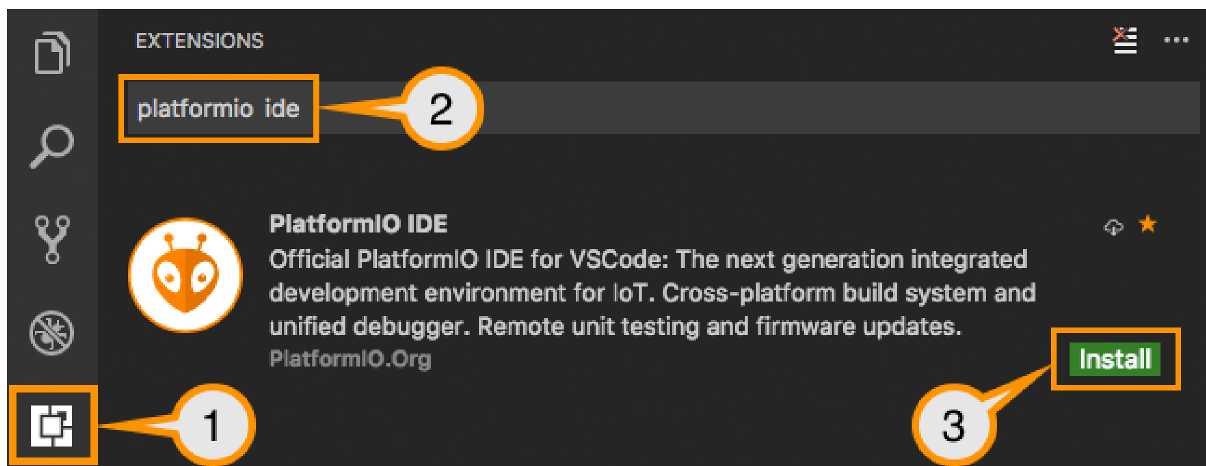
- 支持的硬件平台：Atmel AVR, Atmel SAM, Espressif 32, Espressif 8266, Freescale Kinetis, Intel ARC32, Lattice iCE40, Maxim 32, Microchip PIC32, Nordic nRF51, Nordic nRF52, NXP LPC, Silicon Labs EFM32, ST STM32, Teensy, TI MSP430, TI Tiva, WIZNet W7500 等；
- 支持的嵌入式架构：Arduino, ARTIK SDK, CMSIS, Energia, ESP-IDF, libOpenCM3, mbed, Pumbaa, Simba, SPL, STM32Cube, WiringPi 等；
- 支持的代码编辑器：Atom, CLion, VS Code, Eclipse, Emacs, Vim 等。

相对于 Arduino IDE，使用 PlatformIO IDE 可以实现代码的自动补全提示和代码纠错，而且有着更快的编译速度。PlatformIO IDE 同样支持 Arduino IDE 的库管理，是目前开发 Arduino 程序和其他嵌入式程序的最佳选择。

3.3.2 PlatformIO IDE 安装

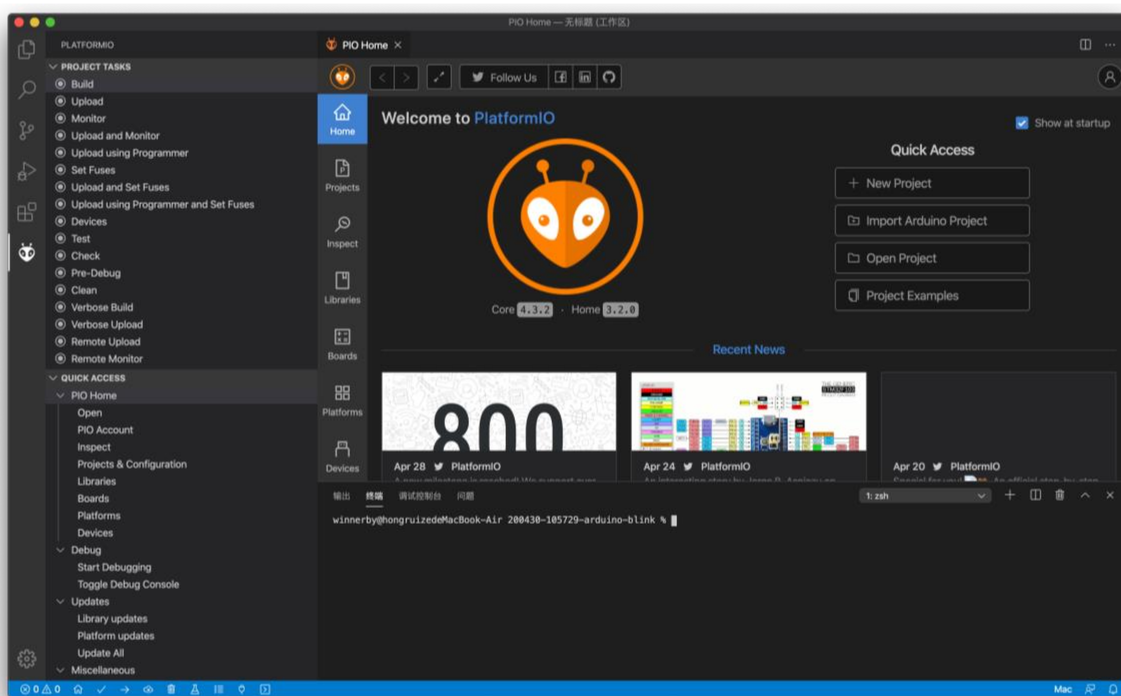
由于 PlatformIO Core 由 Python 编写，同时 PlatformIO IDE for VSCode 需要依赖 Visual Studio Code，所以首先需要安装 VSCode 和 Python 3.5+。

打开 VSCode，点击左侧的扩展选项卡，然后在搜索栏中搜索“PlatformIO IDE”，点击安装。



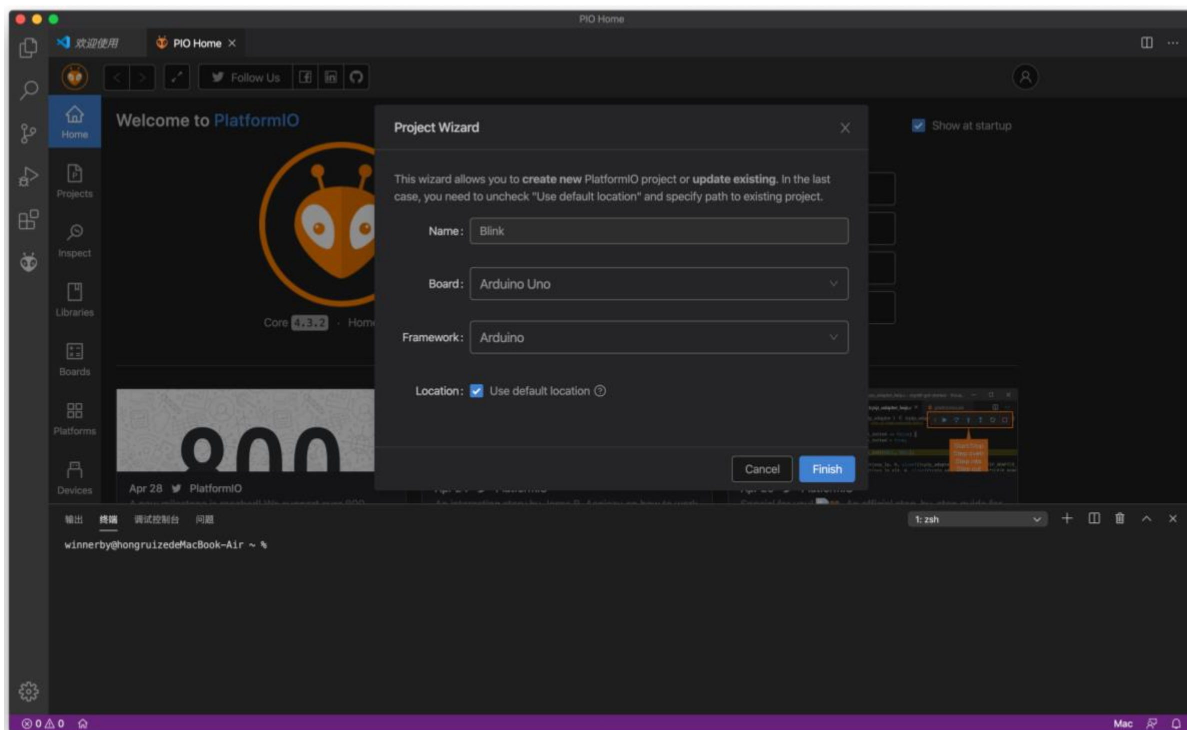
安装过程中需要进行一次重新加载 VSCode 操作，当弹出提示框要求重新加载时，点击 Reload 即可，重新加载后会自动继续进行安装，等待安装结束即可。

安装后，点击 VSCode 左侧新增的 PlatformIO 进入 PlatformIO IDE 页面，可以创建工程、打开示例工程等。



3.3.3 创建 Arduino 工程

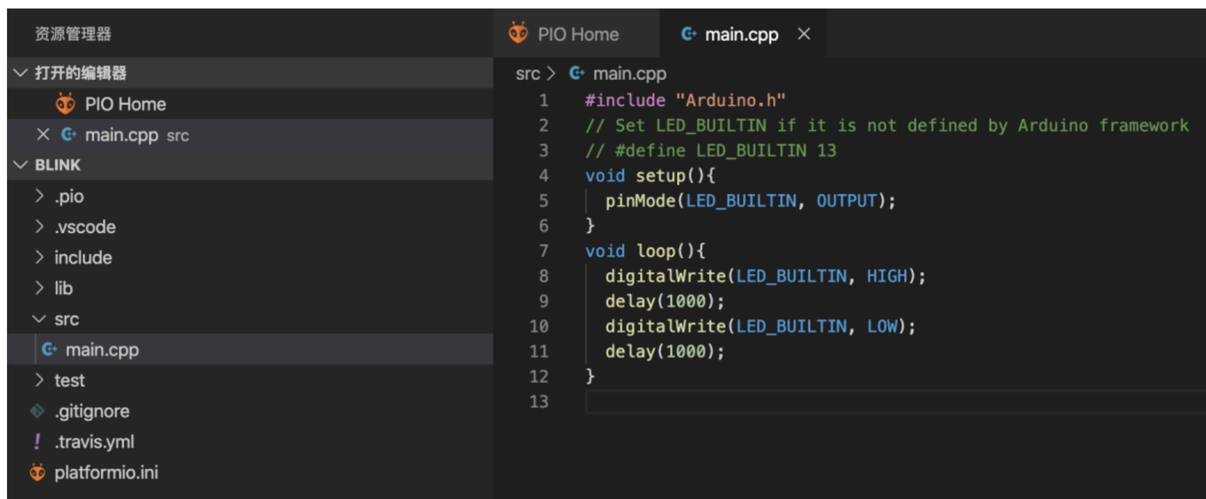
1、点击 PIO Home 页面的“New Project”，设置工程名称，选择开发板名称和框架，即可新建工程。工程默认路径可以将鼠标悬停在 Location 处获得，取消勾选可以自定义工程路径。



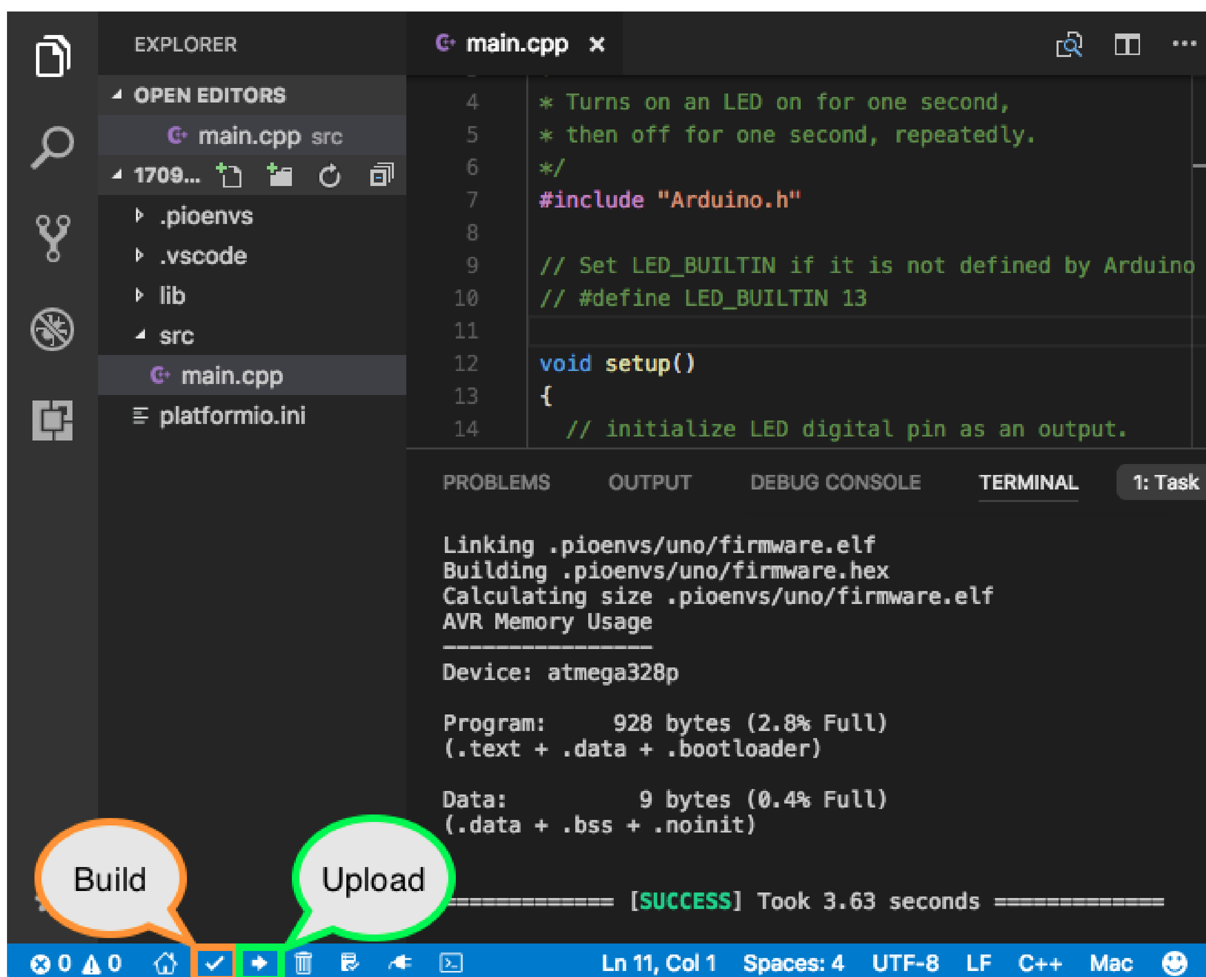
第一次创建 Arduino 工程时，PlatformIO IDE 将会下载必要的组件，之后再次创建工程时耗费时间会大幅度缩短。

2、点击工程的 src 文件夹中的 main.cpp 文件，将其中内容替换为：

```
#include "Arduino.h"
// Set LED_BUILTIN if it is not defined by Arduino framework
// #define LED_BUILTIN 13
void setup(){
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop(){
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

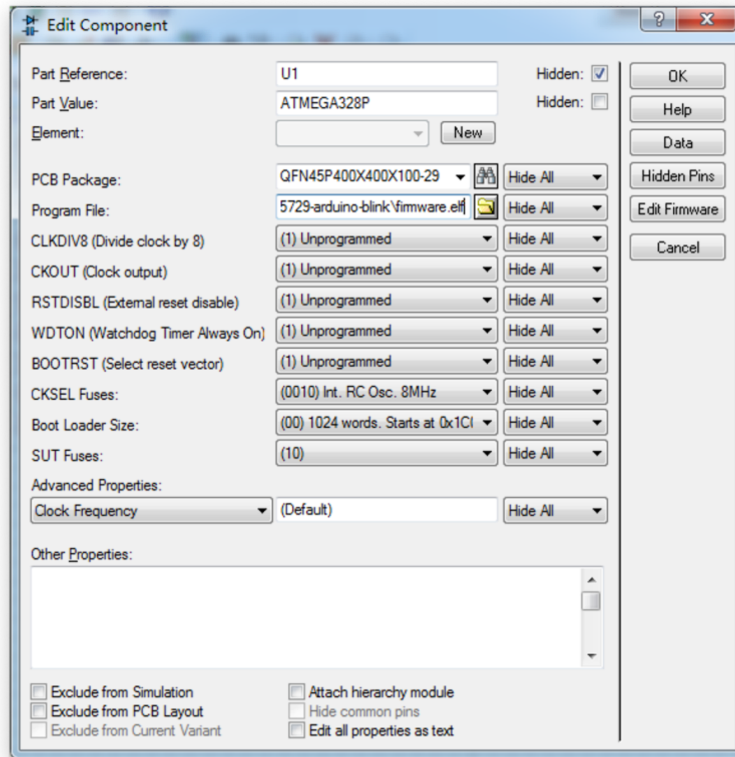


3、通过快捷键 Ctrl + Alt + B 或者通过 VSCode 下方 PlatformIO 工具栏的“Build”按钮进行编译。编译生成的 ELF 程序文件路径在输出中即会体现。新版本 PlatformIO 的 ELF 文件位于工程的/.pio/build/uno 文件夹下。编译会同时产生 ELF 文件和 HEX 文件，这两种程序文件均可以在 Proteus 中进行仿真。

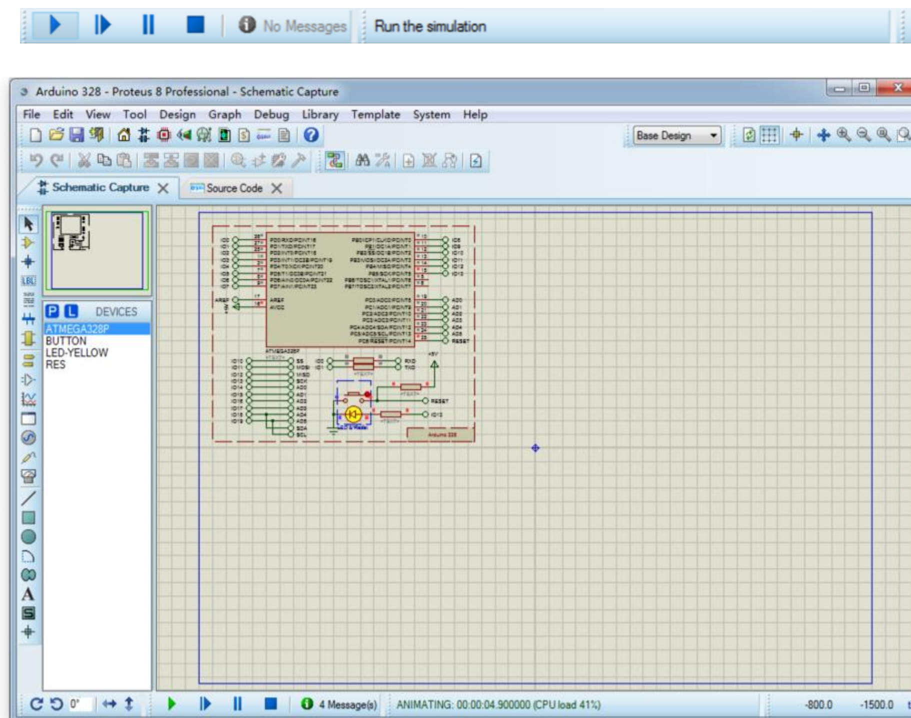


3.4 仿真 Arduino 程序

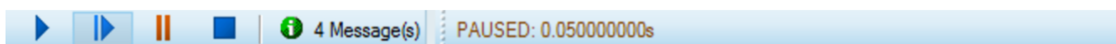
1、载入程序文件。



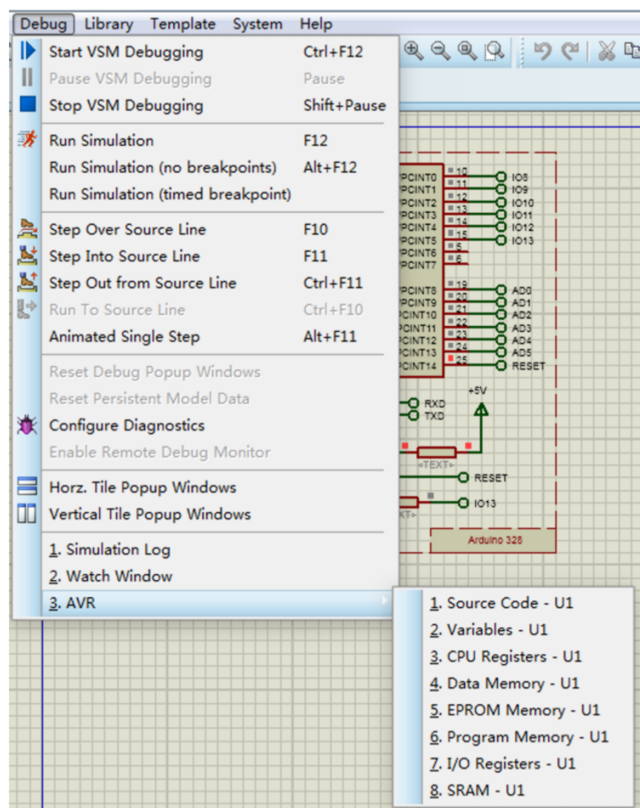
2、点击下方仿真工具栏的 Run，可以看到仿真已经运行，LED 闪烁。



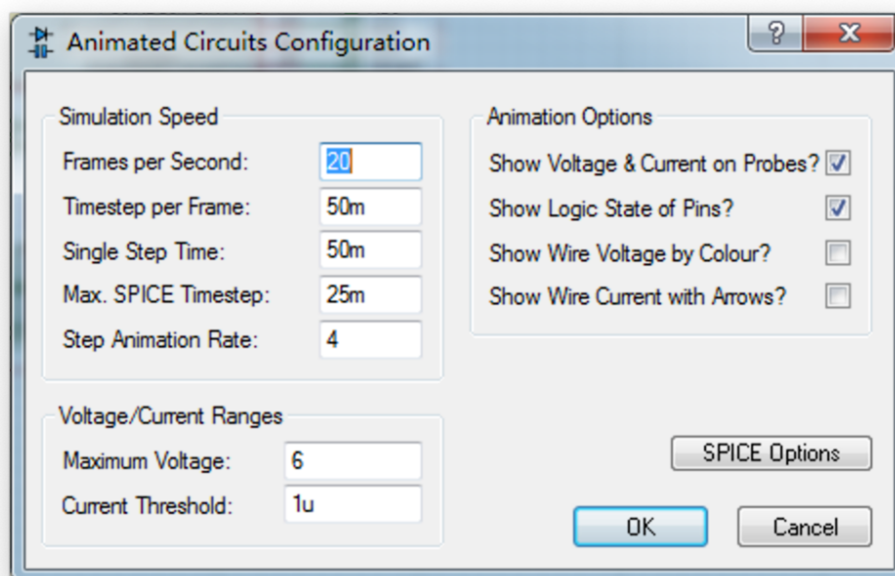
2、点击下方仿真工具栏的第二个按钮可以单步运行（即 0.05s）。



3、点击菜单栏中的“Debug”□“AVR”可以查看寄存器、IO 等信息。



4、在菜单栏的“System”□“Set Amination Options”可以设置仿真速度。



第二篇 基础实验篇

控制是嵌入式开发不可少的部分，本篇将通过 7 个基础实验，介绍 Arduino 的各类外部接口。

本篇分为如下 7 个章节：

第 4 章 流水灯实验

第 5 章 蜂鸣器实验

第 6 章 按键输入实验

第 7 章 虚拟串口实验

第 8 章 外部中断实验

第 9 章 定时器中断实验

第 10 章 LCD1602 实验

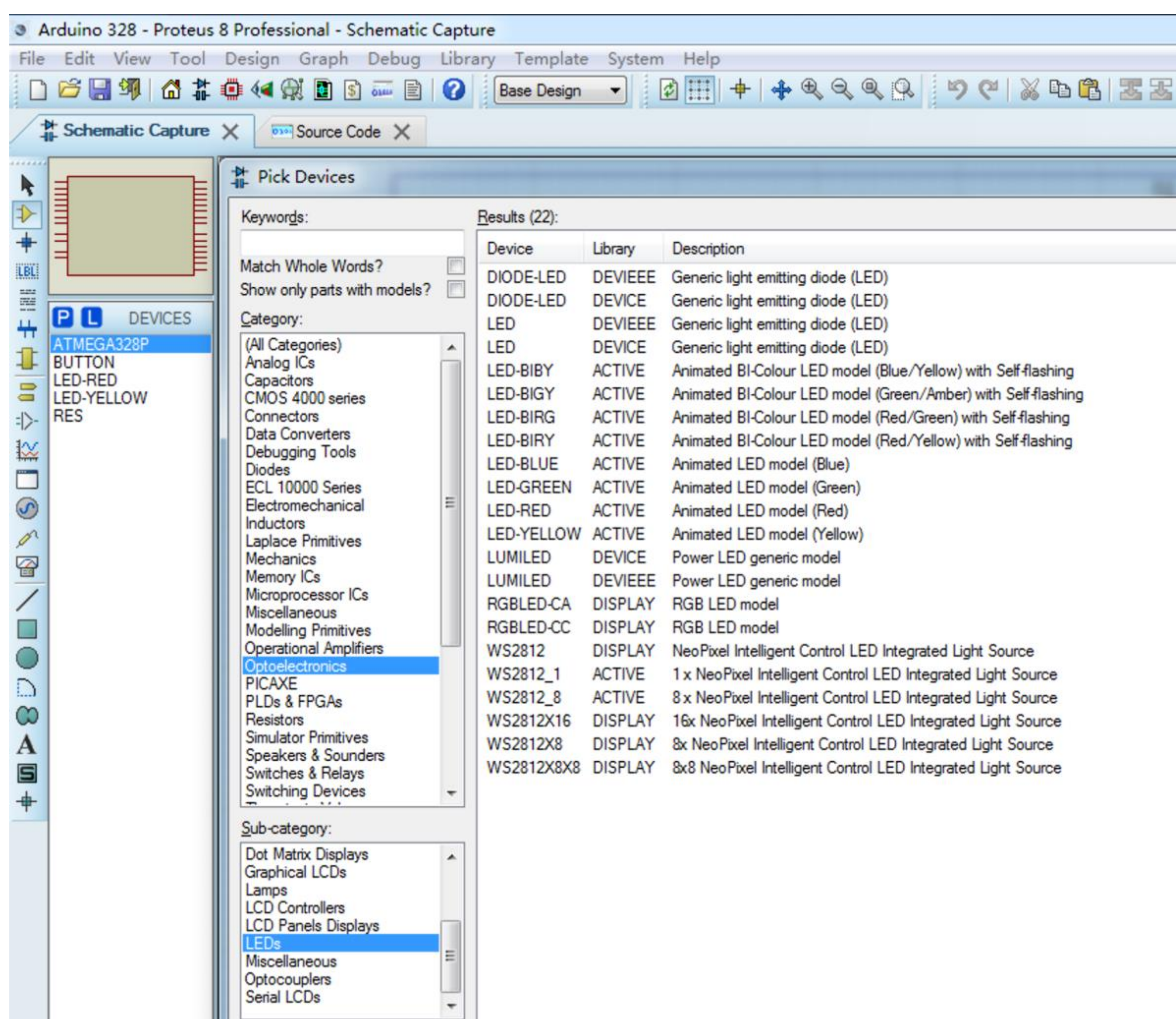
第4章　流水灯实验

本章将要通过数字 IO 端口实现一个流水灯的效果。通过这一章的学习，你将初步掌握 Arduino 数字 IO 端口的使用，而这是面向 Arduino 开发的第一步。

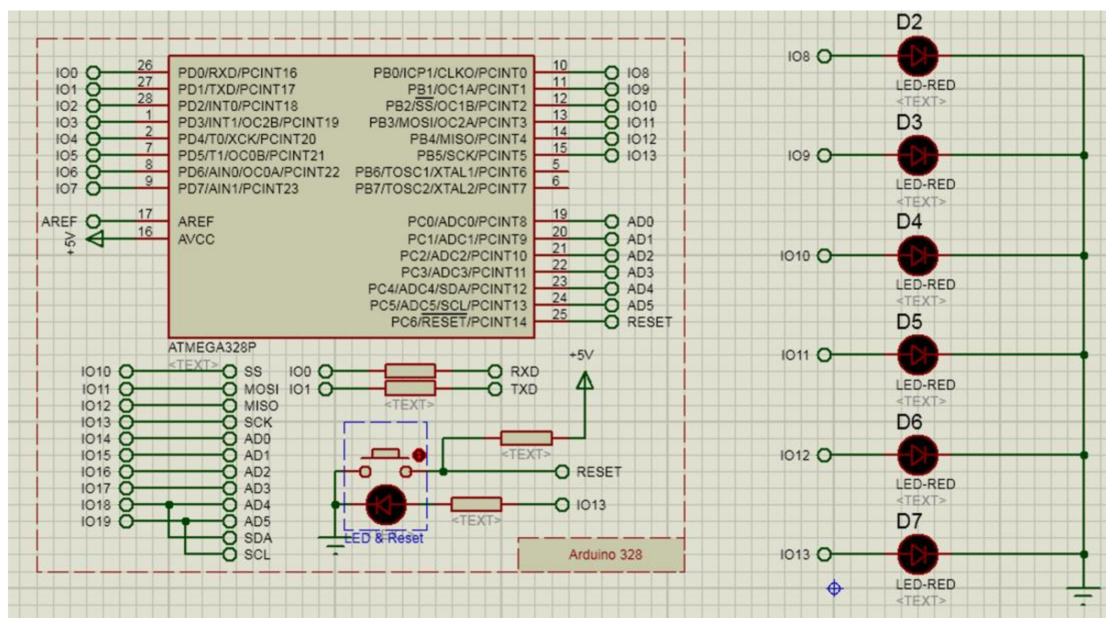
4.1　硬件设计

本章实验用到的硬件只有 LED 灯。

点击 Proteus 左侧元件栏的“P”按钮，打开 Pick Devices 窗口。在 Category 中找到 Optoelectronics，选择 LEDs，可以看到有四种颜色的单色 LED 灯，可以根据喜好自行组合。



LED 可以选择共阴极或共阳极连接方式，共阴极就是将 LED 的负极统一接到 GND，而正极分别接到 Arduino 的数字引脚上。这里将其接入到 IO8～IO13 上。标签可以通过左边栏“Terminals Mode”选择，双击设定。



4.2 软件设计

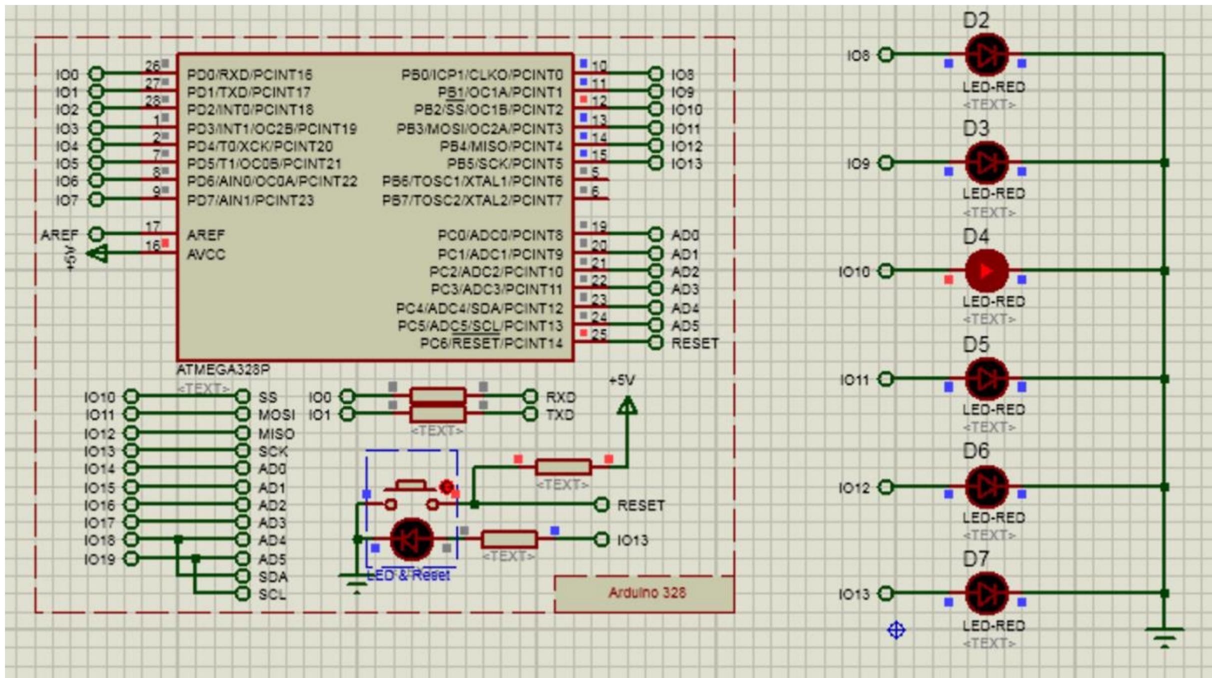
Arduino 程序主要分为两部分：setup()和 loop()。setup()主要是对端口的配置，包括端口初始化和端口初始状态，只执行一次；loop()是 Arduino 程序的主体，并循环执行。

流水灯实验代码如下：

```
#include <Arduino.h>

int led_pin[] = {8, 9, 10, 11, 12, 13};     // LED 引脚

int led_cnt = 6;      // LED 数量

void setup() {

  for(int i=0; i<led_cnt; i++){

    pinMode(led_pin[i], OUTPUT);      // 设置 LED 引脚为输出模式

    digitalWrite(led_pin[i], LOW);   // 设置输出低电平

  }

}

void loop() {

  for(int i=0; i<led_cnt; i++){

    digitalWrite(led_pin[i], HIGH);     // 设置输出高电平

    delay(200);                          // 延时 200ms

    digitalWrite(led_pin[i], LOW);      // 设置输出低电平

  }

}
```

4.3 仿真测试

将程序文件加载到 Protues 中，执行仿真，LED 灯流动闪烁。



第5章　蜂鸣器实验

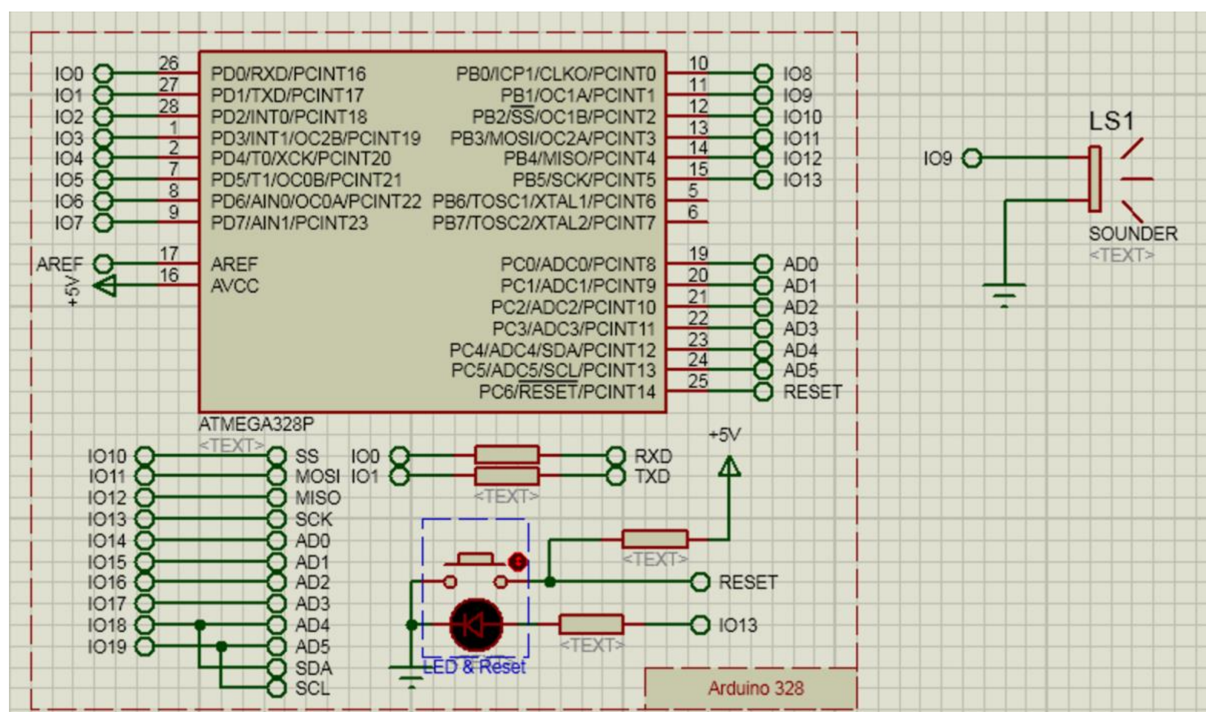
本章将要通过数字 IO 端口输出 PWM 实现驱动无源蜂鸣器的效果。通过这一章的学习，你将初步掌握 Arduino 数字 IO 端口输出 PWM 的使用。

5.1　硬件设计

本章实验用到的硬件只有蜂鸣器。蜂鸣器分为有源蜂鸣器和无源蜂鸣器两种，其中有源蜂鸣器可以直接用直流驱动，而无源蜂鸣器需要脉冲信号才能驱动，且根据脉冲的占空比不同可以输出不同的音调。脉宽调节即为 PWM。

Arduino Uno 的数字 IO3、5、6、9、10、11 均可用作 PWM 输出，PWM 输出使用 analogWrite(pin, dutyCycle)实现，其中 dutyCycle 即为占空比，取值范围为 0～255，对应占空比 0%～100%（127 对应 50%）。但这种方式输出的 PWM 信号的频率是固定的，为 16MHz/64/256，约为 1kHz。

Proteus 可以仿真有源蜂鸣器和无源蜂鸣器。其中，有源蜂鸣器为 BUZZER，默认驱动电压为 12V，可以手动调节；无源蜂鸣器有 SOUNDER 和 SPEAKER 两种，需使用 PWM 驱动。这里选择无源蜂鸣器 SOUNDER，并将其接入到 IO9 上。



5.2 软件设计

本实验需要通过数字 IO9 端口输出 PWM 波，同样需要先设置 IO9 为输出模式，然后利用 PWM 输出函数进行输出。实验输出 PWM 的占空比由 0%到 100%逐渐变化。

蜂鸣器实验代码如下：

```
#include <Arduino.h>
int buzzer_pin = 9;
void setup() {
  pinMode(buzzer_pin, OUTPUT);
}
void loop() {
  for(int i=0; i<=255; i++){
    analogWrite(9, i);
    delay(50);
  }
}
```

5.3 仿真测试

蜂鸣器在仿真时通过电脑声卡进行输出，可以听到蜂鸣器音调逐渐升高。在单步运行模式下更容易听出音调的差别。

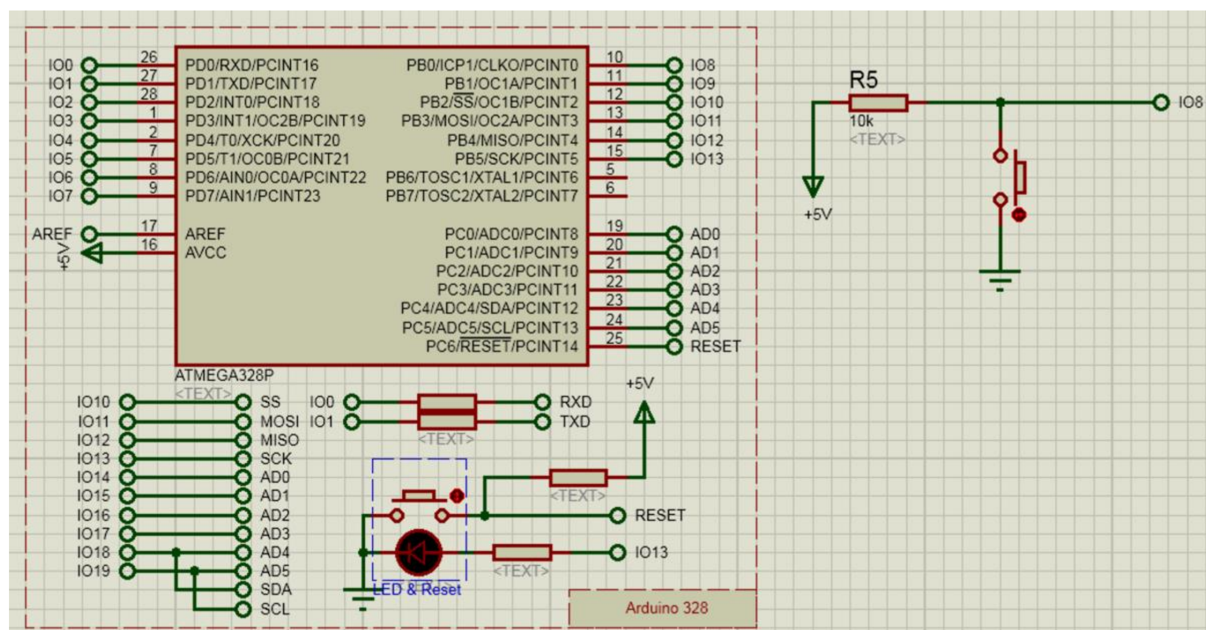
另外输出的 PWM 信号可以用 Proteus 中的示波器进行观察。

第6章　按键输入实验

6.1　硬件设计

上两章讲了数字 IO 的两种输出，本章利用按键学习数字 IO 的输入。

硬件上，使用一个按键作为数字 IO 的输入，同时利用板载的 LED 灯进行实验效果展示。为了避免引脚电平悬空，通常选用上拉电阻接法或下拉电阻接法。这里选用上拉电阻接法，用上拉电阻将引脚电平钳制在高电平，按键按下后变为低电平。



6.2　软件设计

同样的，需要先对 IO8 配置为输入，再进行按键读取。

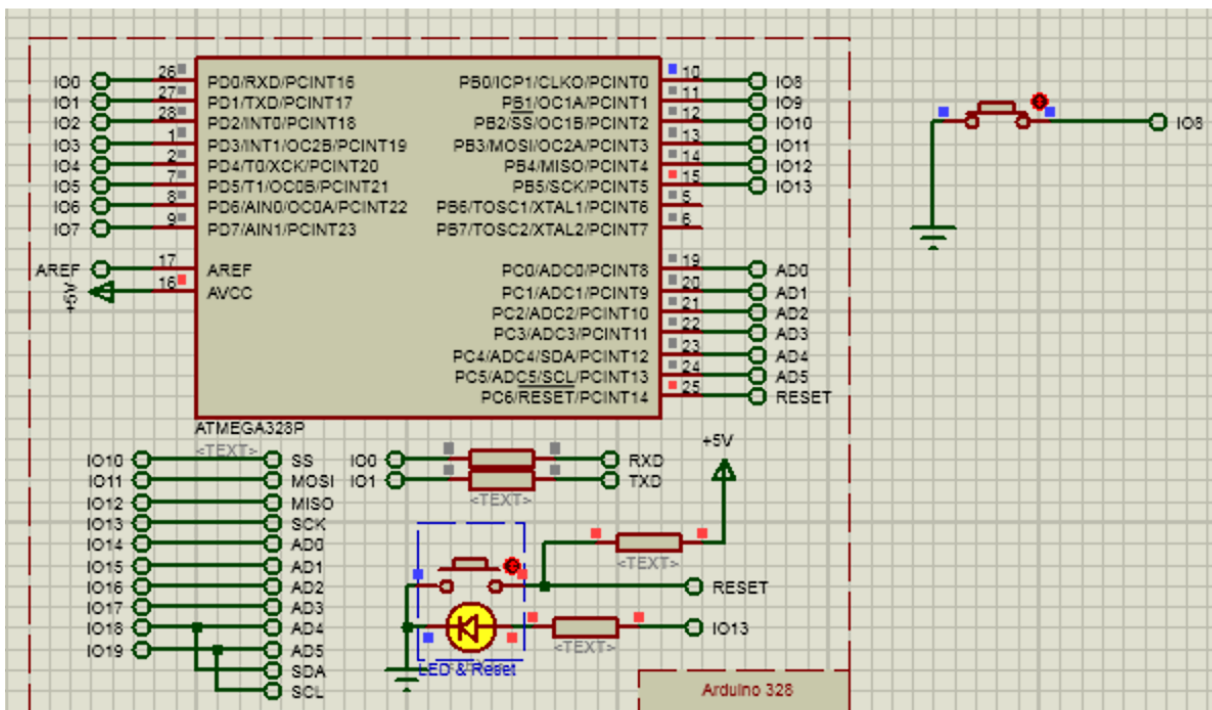
按键输入实验代码如下：

```
#include <Arduino.h>

int key_pin = 8;

void setup() {

  pinMode(key_pin, INPUT);

  pinMode(LED_BUILTIN, OUTPUT);

  digitalWrite(LED_BUILTIN, LOW);

}

void loop() {

  int key_up = 1;     //  按键松开标志
```

```
        if(key_up && digitalRead(key_pin)==0){
          delay(10);          // 消除抖动
          key_up = 0;
          if(digitalRead(key_pin)==0){
            digitalWrite(LED_BUILTIN, HIGH);
          }
          else if(digitalRead(key_pin)==1){
            digitalWrite(LED_BUILTIN, LOW);
            key_up = 1;
          }
        }
      }
```

6.3 仿真测试

按下按键，LED 亮；松开按键，LED 灭。

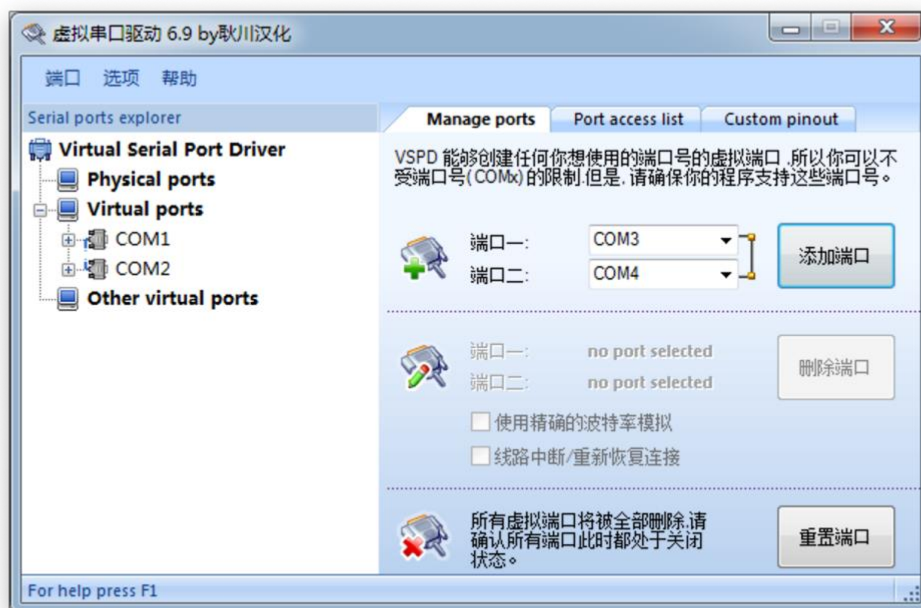


第7章 虚拟串口实验

7.1 硬件设计

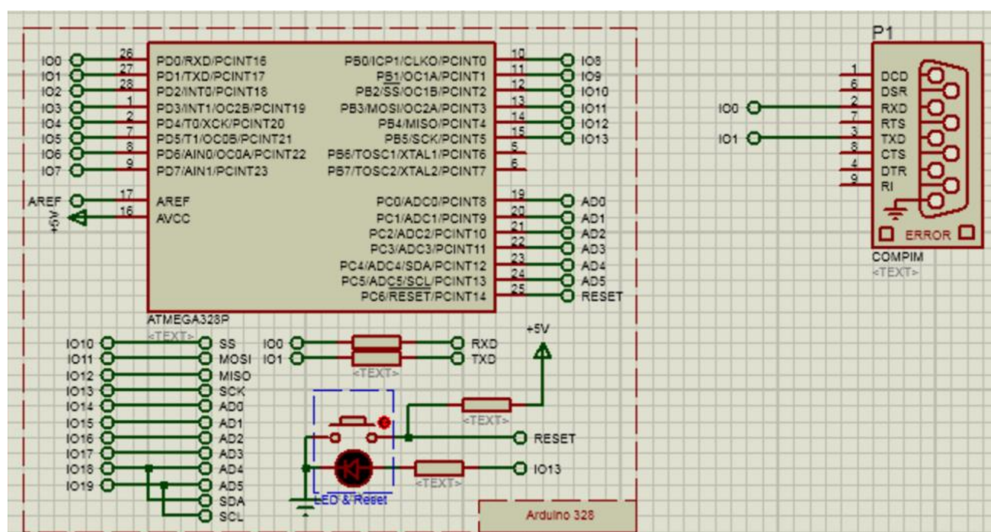
串口是 Arduino 的重要外部接口，同时也是软件开发时重要的调试手段。Proteus 提供了虚拟串口，可以用作各类 MCU 的串口仿真。

首先，需要安装虚拟串口工具，安装后创建一对虚拟串口。

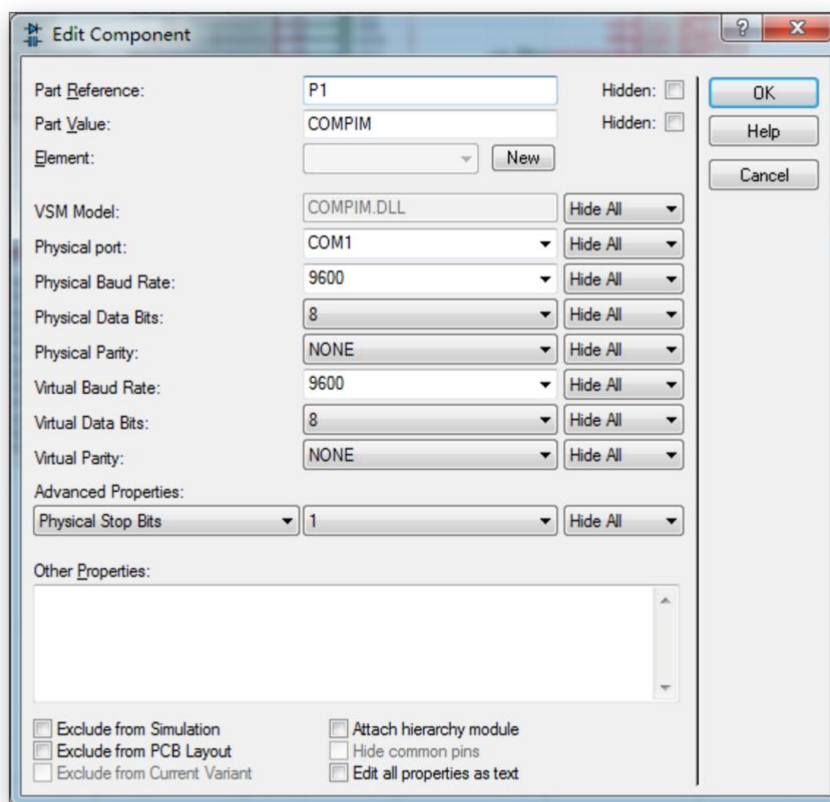


COM1 和 COM2 即为创建的虚拟串口。其中一个连接到 Proteus，另一个连接到串口调试助手。

硬件设计上，Arduino 的 IO0 作为 RXD 接收引脚，IO1 作为 TXD 发送引脚，在元件库中找到“COMPIM”，并将 RXD 和 TXD 对应连接起来。



双击 COMPIM，配置串口为 COM1，波特率 9600，数据位 8，无校验位，停止位 1。



7.2 软件设计

首先要对串口进行初始化，设定其波特率。串口通过 Serial.read()函数接收一个字符，通过 Serial.print() 和 Serial.println() 函数发送数据。Serial.available()函数用于判断串口缓冲区状态，并返回缓冲区存在的字节数。

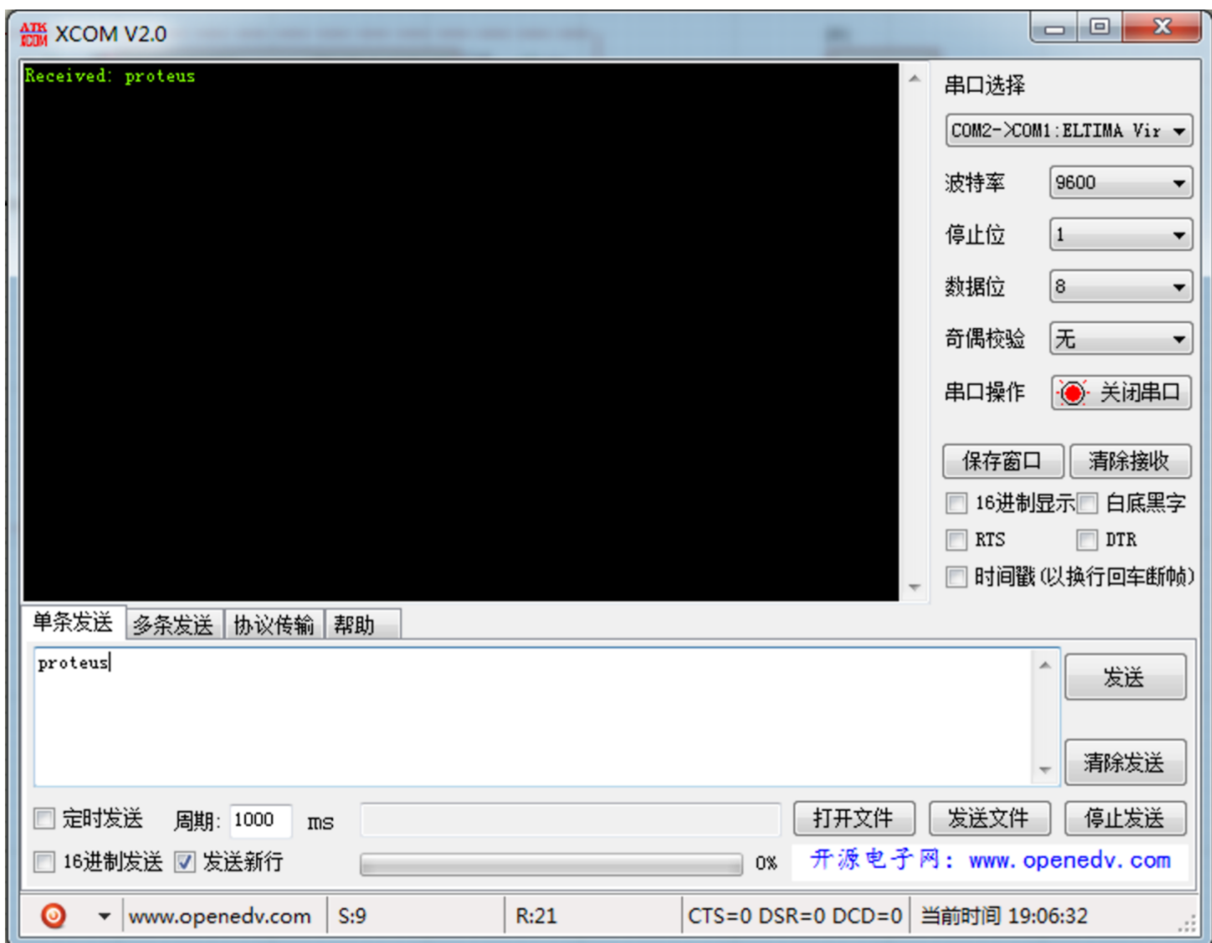
虚拟串口实验代码如下：

```
#include <Arduino.h>

String comdata = "";

void setup(){

    Serial.begin(9600); // 初始化串口，波特率为 9600

}

void loop(){

    while (Serial.available() > 0){

        comdata += char(Serial.read());

        delay(2);
```

```
        }
        if (comdata.length() > 0){
            Serial.print("Received: ");
            Serial.println(comdata);
            comdata = "";
        }
    }
```

7.3 仿真测试

加载程序文件后，启动仿真。打开串口调试助手，配置端口号为COM2□COM1，数据位 8，停止位 1，无校验位。通过串口调试助手发送数据，返回发送的数据内容，如下图所示。



第8章　外部中断实验

8.1　硬件设计

Arduino Uno 的数字 IO2、IO3 可以作为外部中断的输入。外部中断共有四种触发方式：低电平触发（LOW）、电平变化触发（CHANGE）、上升沿触发（RISING）和下降沿触发（FALLING）。

本章利用外部中断实现第 6 章按键输入实验的实验效果，硬件连接与第 6 章类似，只不过需要将按键连接到支持外部中断的数字 IO2 引脚上。

8.2　软件设计

Arduino Uno 的外部中断只需要在 setup()中配置外部中断，并单独编写中断服务函数即可。配置外部中断的函数如下：

attachInterrupt(interrupt, function, mode);

其中，interrupt 为中断通道编号，IO2 对应 INT0，IO3 对应 INT1；function 为中断服务函数，mode 为中断触发模式。

注意，如果使用 PlatformIO IDE 开发环境，自编写函数需要先声明再定义，否则编译时会报错。如果使用 Arduino IDE 则不存在这个问题。

外部中断实验代码如下：

```
#include <Arduino.h>
volatile byte state = LOW;
void led_on();
void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
  attachInterrupt(INT0, led_on, CHANGE);
}
void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(LED_BUILTIN, state);
}
void led_on() {
  state = !state;
}
```

8.3 仿真测试

实验现象与第 6 章按键输入实验相同，按下按键，LED 亮；松开按键，LED 灭。

第9章　定时器中断实验

9.1　硬件设计

本章讲定时器中断前，先介绍 Arduino Uno 的定时器。

Arduino Uno 共有三个定时器 timer0、timer1、timer2，其中：

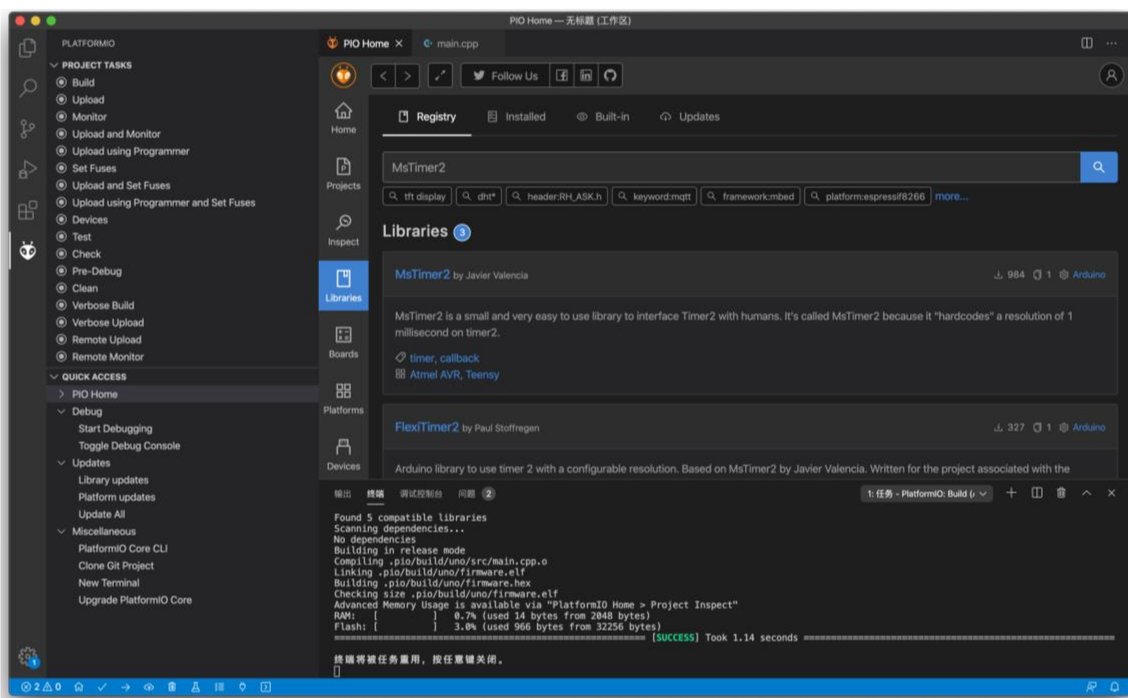
timer0 负责 IO5、IO6 的 PWM 输出，同时负责 millis()函数和 delay()函数的计时，所以 IO5、IO6 输出的 PWM 精度比其他引脚略低；

timer1 负责 IO9、IO10 的 PWM 输出，此外，Arduino 的伺服电机库 Servo 也需要用到 timer1；

timer2 负责 IO3、IO11 的 PWM 输出。

如果要使用定时器作为中断触发，可以使用 TimerOne 库（使用 timer1）或者 MsTimer2 库（使用 timer2）。需要注意的是，当设置某个定时器为中断时，其控制的 PWM 引脚无法进行 PWM 输出。

本章选用 MsTimer2 库，在 Arduino IDE 中无需其他操作，而在 PlatfromIO IDE 中，需要安装 MsTimer2 库。在 PIO Home 界面选择 Library 选项卡，搜索 MsTimer2，安装即可。



本章硬件上无需连接其他电路，直接采用 Arduino Uno 板载的 LED 灯即可，定时器控制 LED 闪烁。

9.2 软件设计

MsTimer2 库有三个函数，分别是：

MsTimer2::set(unsigned long ms, void (*f)())，定时器中断初始化函数，其中第一个参数计时时间，单位为 ms，第二个参数为中断服务函数；

MsTimer2::start()，定时器中断开使能，即启动定时器中断；

MsTimer2::stop()，定时器中断关使能，即停止定时器中断。

注意，在 PlatformIO IDE 中，添加库之后由于 VS Code 自身不识别库，会认为有语法错误，但是编译可以正常进行，不会报错。

定时器中断实验代码如下：

```
#include <Arduino.h>
#include <MsTimer2.h>
void flash() {
  static boolean output = HIGH;
  digitalWrite(13, output);
  output = !output;
}
void setup() {
  pinMode(13, OUTPUT);
  MsTimer2::set(500, flash); // 500ms period
  MsTimer2::start();
}
void loop() {
}
```

9.3 仿真测试

加载程序文件之后，运行仿真，板载 LED 灯每 500ms 变换一次状态，闪烁周期为 1s。在单步运行模式下可以更好地观察闪烁时间。

第10章 LCD1602 实验

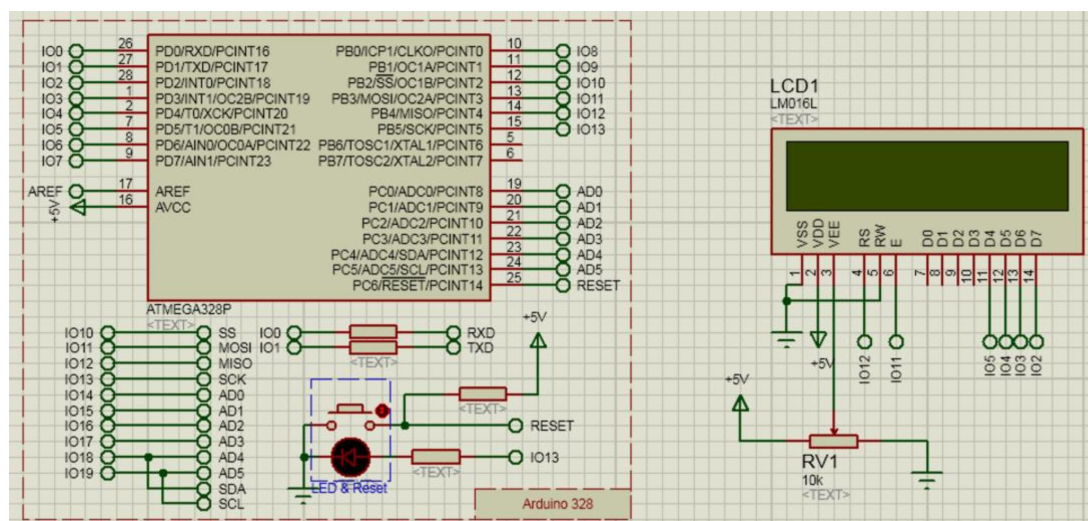
10.1 LCD1602 介绍

LCD1602 是一种工业字符型液晶，能够同时显示 16*02 即 32 个字符。LCD1602 液晶显示的原理是利用液晶的物理特性，通过电压对其显示区域进行控制，即可以显示出图形。LCD1602 引脚如下：

引脚	符号	说明
1	GND	电源地
2	VCC	5V
3	V0	对比度调整
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；2:写）
6	EN	使能，高电平读取信息，负跳变执行指令
7～14	D0～D7	8 位双向数据端
15	BLA	背光正极
16	BLK	背光负极

10.2 硬件设计

Proteus 中 LCD1602 对应模块为 LM016L，并且没有两个背光引脚。另外，实验还需要一个 10k 的电位器 POT。硬件连接如下：



10.3 软件设计

LCD1602 使用 Hitachi HD44780 作为显示控制器，可以使用 Arduino 的

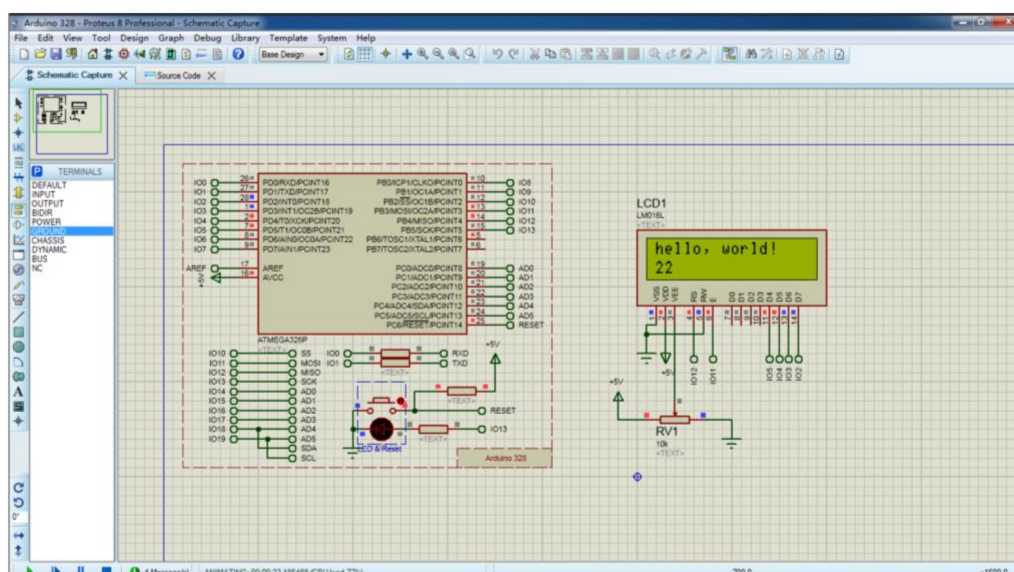
LiquidCrystal 库，这个库适用于 HD44780 芯片及其兼容芯片，依赖 Wire 库。

LCD1602 实验代码如下：

```
#include <Arduino.h>
#include <Wire.h>
#include <LiquidCrystal.h>
// 初始化 LCD 控制引脚
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
void setup() {
  // 初始化 LCD1602 的行数和列数:
  lcd.begin(16, 2);
  // LCD 显示 Hello World
  lcd.print("hello, world!");
}
void loop() {
  // 设置要显示的位置为第二行第一列
  lcd.setCursor(0, 1);
  // 打印当前秒数
  lcd.print(millis() / 1000);
}
```

10.4 仿真测试

LCD1602 第一行打印 Hello World，第二行打印当前秒数。调节电位器可以调整 LCD1602 的对比度。



第三篇 传感器实验篇

感知是物联网设备获取环境信息的重要方式，本篇通过四个传感器实验，介绍 Arduino 传感器的使用。同时，Proteus 还提供其他传感器，位于 Transducers 器件库中。

本篇分为如下 4 个章节：

第 11 章 DHT11 温湿度传感器实验

第 12 章 GP2D12 红外传感器实验

第 13 章 LDR 光敏电阻实验

第 14 章 BMP180 气压传感器实验

第11章 DHT11 温湿度传感器实验

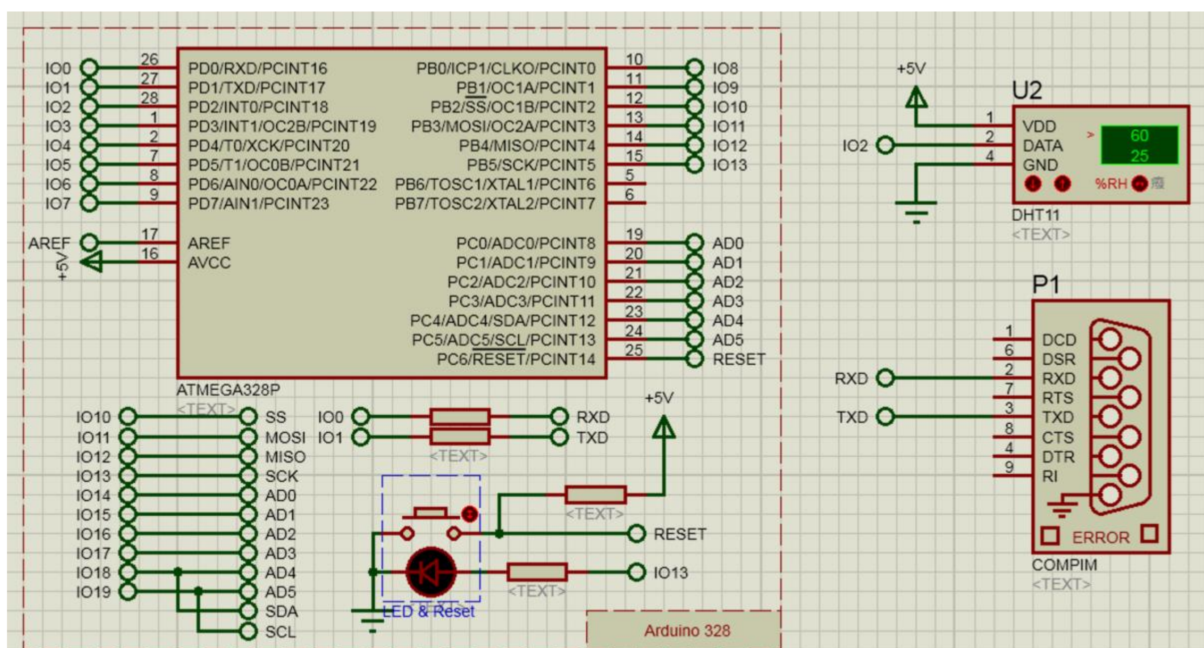
11.1 DHT11 温湿度传感器介绍

DHT11 是一款有已校准数字信号输出的温湿度传感器。其量程精度为温度 0-50°C（±2°C），湿度 20-80%（±5%）。

DHT11 有四根引脚，分别为 VCC（3-5V 电源）、DATA（数据）、NC（悬空）和 GND（电源地）。

11.2 硬件设计

本章实验需要从 DHT11 读取温湿度信息，并通过串口打印出来。硬件连接如下：



11.3 软件设计

DHT11 需要安装 DHT sensor library 这个库，依赖于 Adafruit 库。通过创建 DHT 对象来配置 DHT11 初始化，并通过库函数对温湿度进行读取，同时可以计算出热指数。

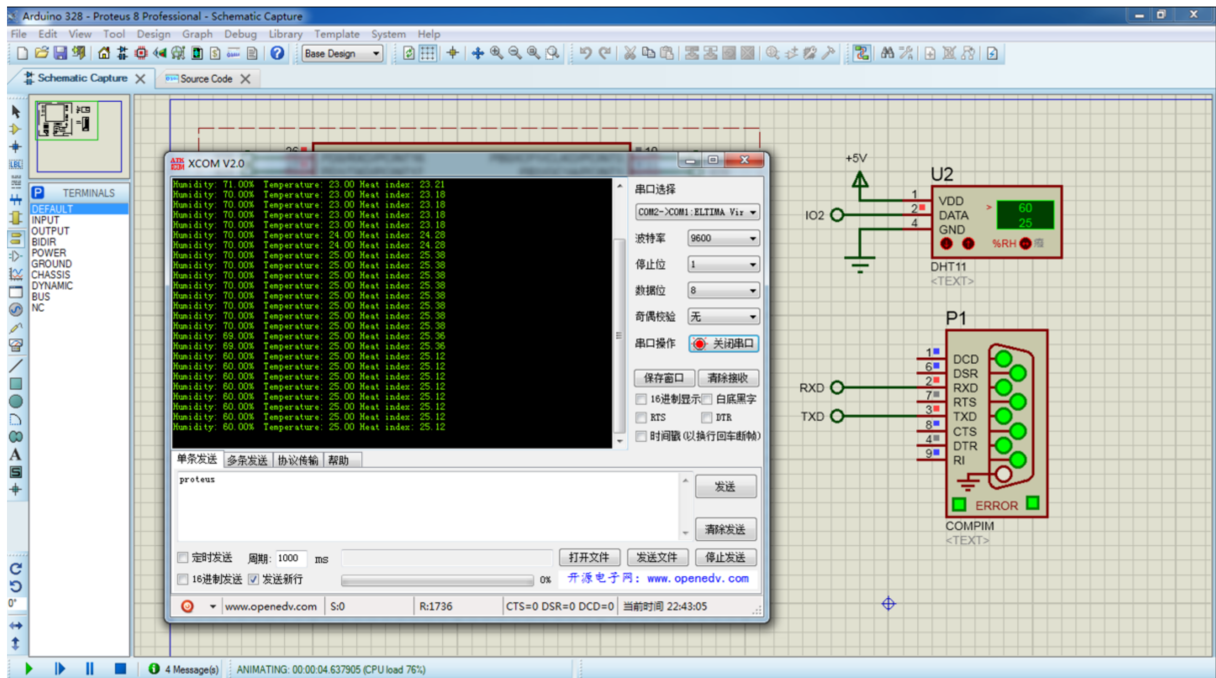
DHT11 温湿度传感器实验代码如下：

```
#include <Arduino.h>
#include <Wire.h>
#include <SPI.h>
```

```
#include <DHT.h>
#define DHTPIN 2        // DHT 数据引脚
#define DHTTYPE DHT11     // DHT11
DHT dht(DHTPIN, DHTTYPE);
void setup() {
  Serial.begin(9600);
  Serial.println(F("DHT11 test!"));
  dht.begin();
}
void loop() {
  delay(1000);   // DHT11 每隔 1s 更新一次数据
  // 读取温湿度
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  // 读取失败处理
  if (isnan(h) || isnan(t)) {
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }
  // 计算热指数（利用摄氏温度）
  float hic = dht.computeHeatIndex(t, h, false);
  Serial.print(F("Humidity: "));
  Serial.print(h);
  Serial.print(F("%   Temperature: "));
  Serial.print(t);
  Serial.print(F(" Heat index: "));
  Serial.println(hic);
}
```

11.4 仿真测试

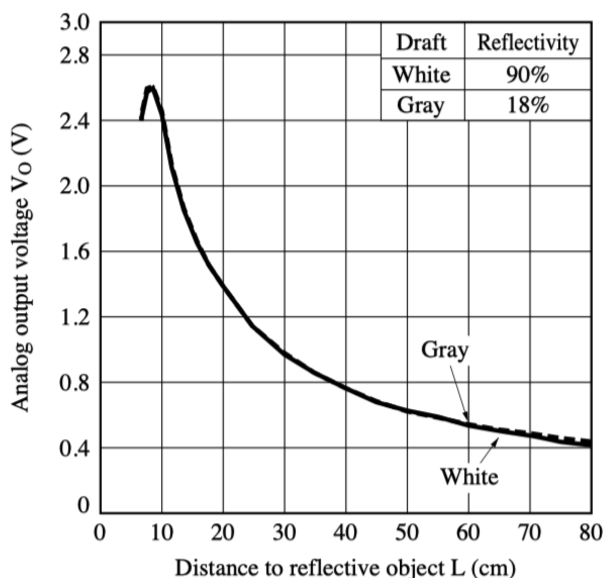
Proteus 中 DHT11 传感器可以设置其温湿度大小，上下键调整数值，左右键选择湿度或温度。串口输出的温湿度信息与 DHT11 面板上设定的温湿度一致，同时输出热指数。



第12章 GP2D12 红外传感器实验

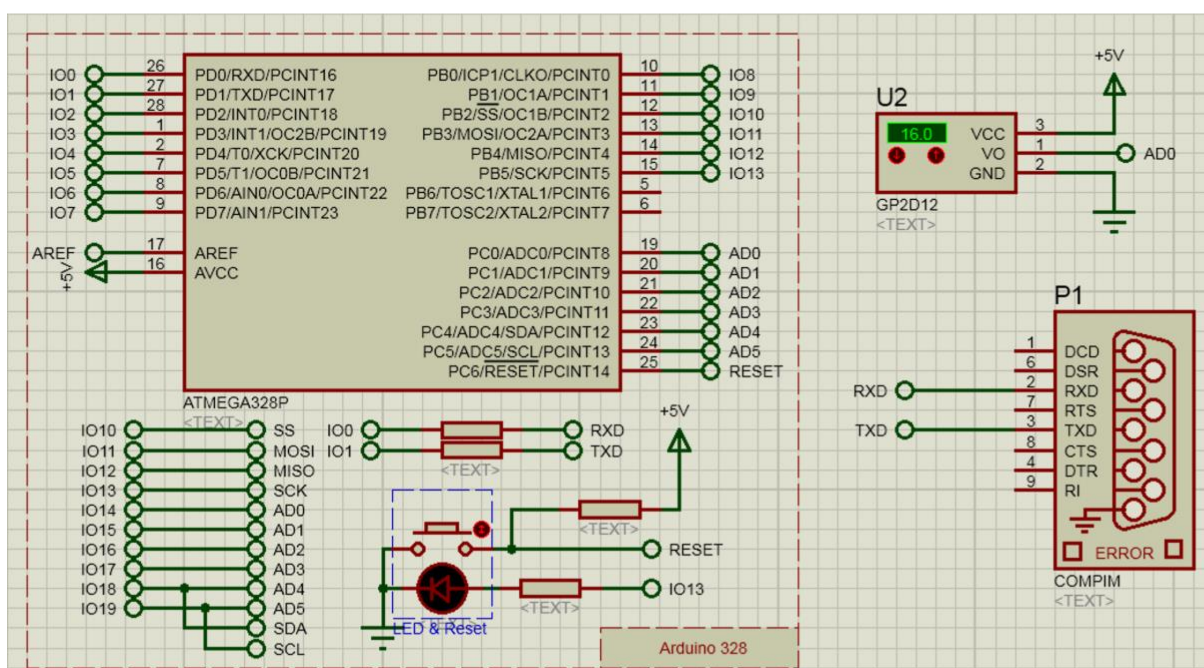
12.1 GP2D12 红外传感器介绍

GP2D12 是 SHARP 公司的一款性价比高、最常用的红外测距传感器。其测量范围为 10-80cm，输出模拟电压为 2.4-0.4V，输出电压与距离成反比非线性关系，如图所示：



12.2 硬件设计

本章实验需要通过模拟 IO 从 GP2D12 读取温湿度信息，并通过串口打印出来。硬件连接如下：



12.3 软件设计

根据 GP2D12 的输出曲线，结合 Arduino 的 ADC 精度、ADC 电压标准等参数，计算出实际距离与采样数据之间的大致关系式为：

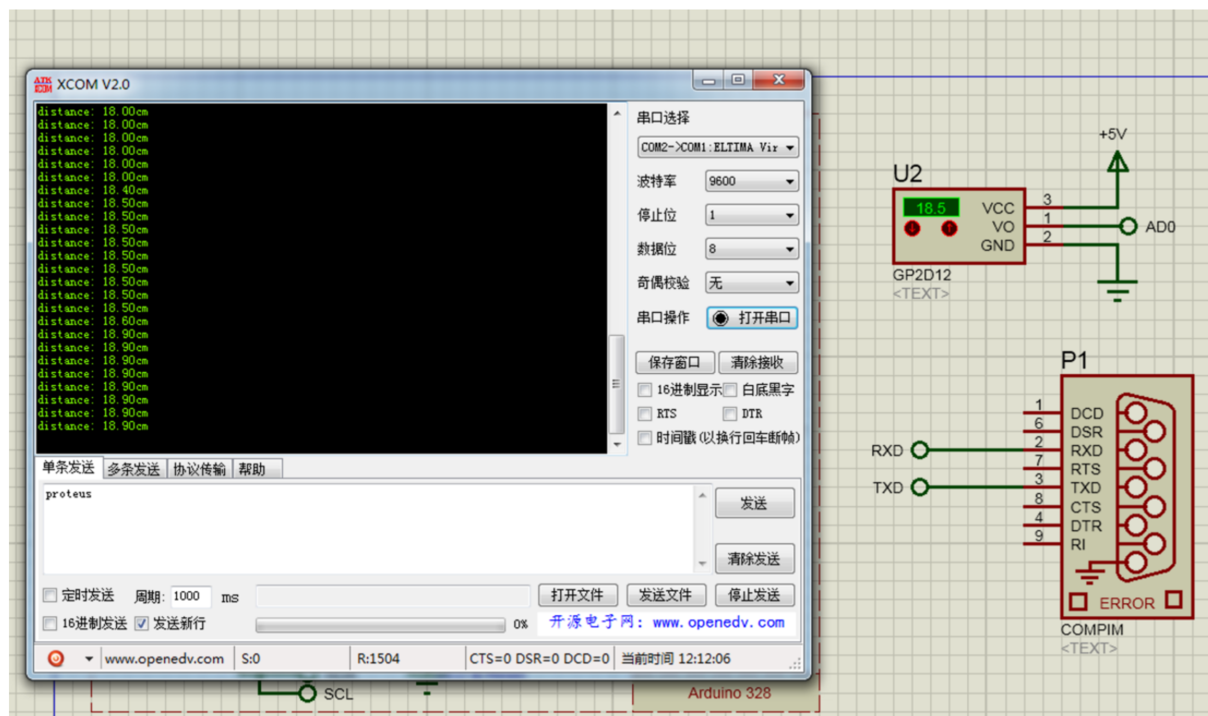
实际距离=2547.8/((float)采样数据*0.49-10.41)-0.42

GP2D12 红外传感器实验代码如下：

```
#include <Arduino.h>

const int GP2D12_PIN = A0;

int val;

float distance;

void setup() {

  Serial.begin(9600);

}

void loop() {

  delay(500);

  val = analogRead(GP2D12_PIN);

  distance = 2547.8/((float)val*0.49-10.41)-0.42;

  if ( distance > 80 || distance < 10 ) {

    Serial.println("ERROR: Over Range!");

  }

  distance = int(distance * 10)/10.0;

  Serial.print("distance: ");

  Serial.print(distance);

  Serial.println("cm");

}
```

12.4 仿真测试

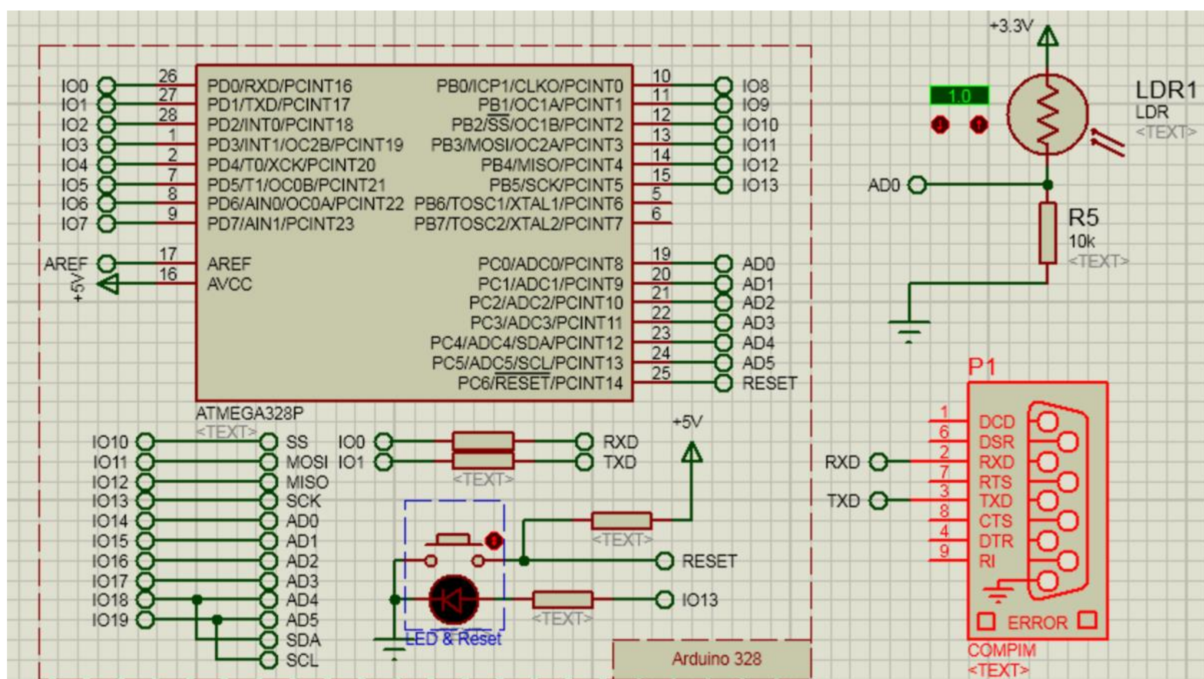
由于 GP2D12 红外传感器输出为模拟电压，且输出与距离成反比非线性关系，故计算得出的距离会有一定的误差，可以根据误差自行调节计算式。在 Protues 中，双击 GP2D12 模块可以设置其调节精度，然后通过面板上的上下键进行距离调节。


XCOM V2.0
distance: 18.00cm
distance: 18.40cm
distance: 18.50cm
distance: 18.60cm
distance: 18.90cm
串口选择
COM2->COM1:ELTIMA Vir
波特率
9600
停止位
1
数据位
8
奇偶校验
无
串口操作
打开串口
保存窗口
清除接收
16进制显示
白底黑字
RTS
DTR
时间戳(以换行回车断帧)
单条发送
多条发送
协议传输
帮助
proteus
发送
清除发送
定时发送
周期:
1000
ms
打开文件
发送文件
停止发送
16进制发送
发送新行
0%
开源电子网: www.openedv.com
www.openedv.com
S:0
R:1504
CTS=0 DSR=0 DCD=0
当前时间 12:12:06
U2
18.5
VCC
VO
GND
GP2D12
<TEXT>
+5V
AD0
P1
DCD
DSR
RXD
RTS
TXD
CTS
DTR
RI
ERROR
COMPIM
<TEXT>
RXD
TXD
SCL
Arduino 328

第13章 LDR 光敏电阻实验

13.1 硬件设计

LDR（Light Dependent Resistor）是一种特殊类型的电阻，阻值与光照强度有关，当光照强的时候表现为低阻值，当光线弱时表现为高阻值。本实验使用一个 10k 电阻和 LDR 搭建分压电路，将 LDR 变化的阻值转换成变化的电压，通过模拟 IO 读取并处理光照数据。



13.2 软件设计

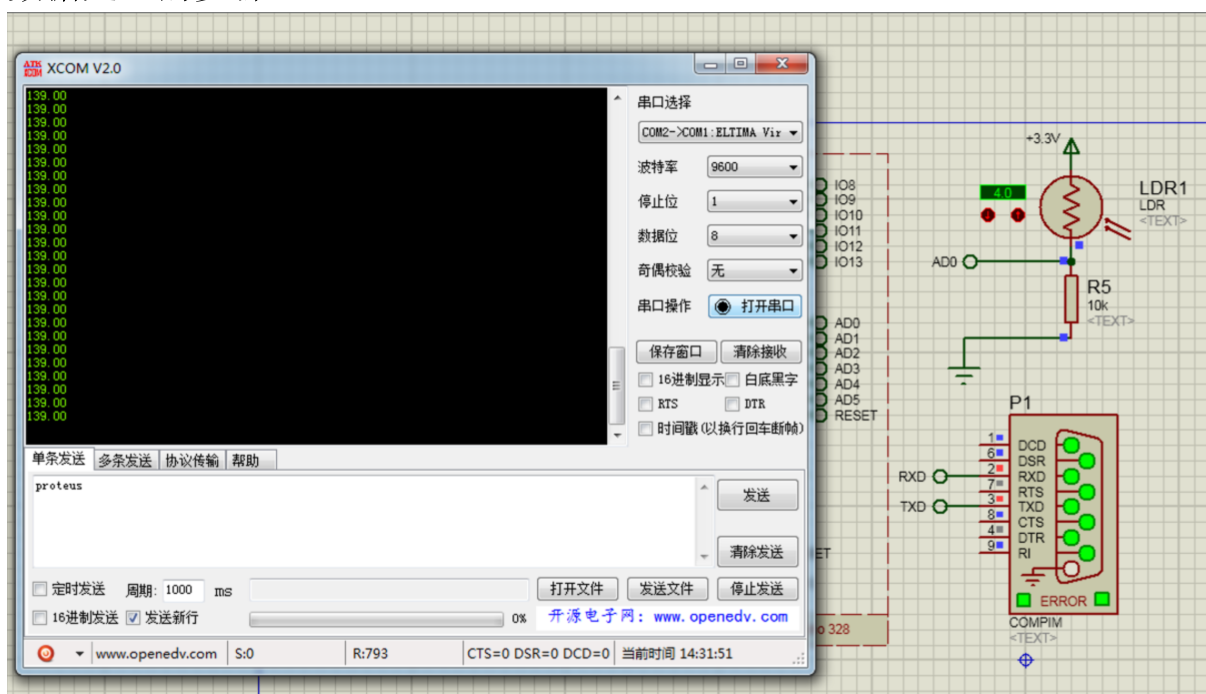
本章实验同样使用模拟 IO 读取电压值，实际上读取的是分压电阻两端的电压值，光强越大则 ADC 读取的值也越大。本章实验代码仅为从串口输出模拟电压值，至于具体如何计算得出实际的光照强度值还需自己编写。实验代码如下：

```
#include <Arduino.h>

const int LDR_PIN = A0;

float var;

void setup() {

  Serial.begin(9600);

}

void loop() {
```

```
        delay(500);
        var = analogRead(LDR_PIN);
        Serial.println(var);
    }
```

13.3 仿真测试

实验通过串口输出读取的 ADC 值，可以发现该值与光照强度（lux）大致成正比关系，可以通过计算获取大致的函数表达式。同时，若要进行智能控制，在设定光照强度后，可以通过此实验查询对应的 ADC 值，省去关于数据处理的步骤。



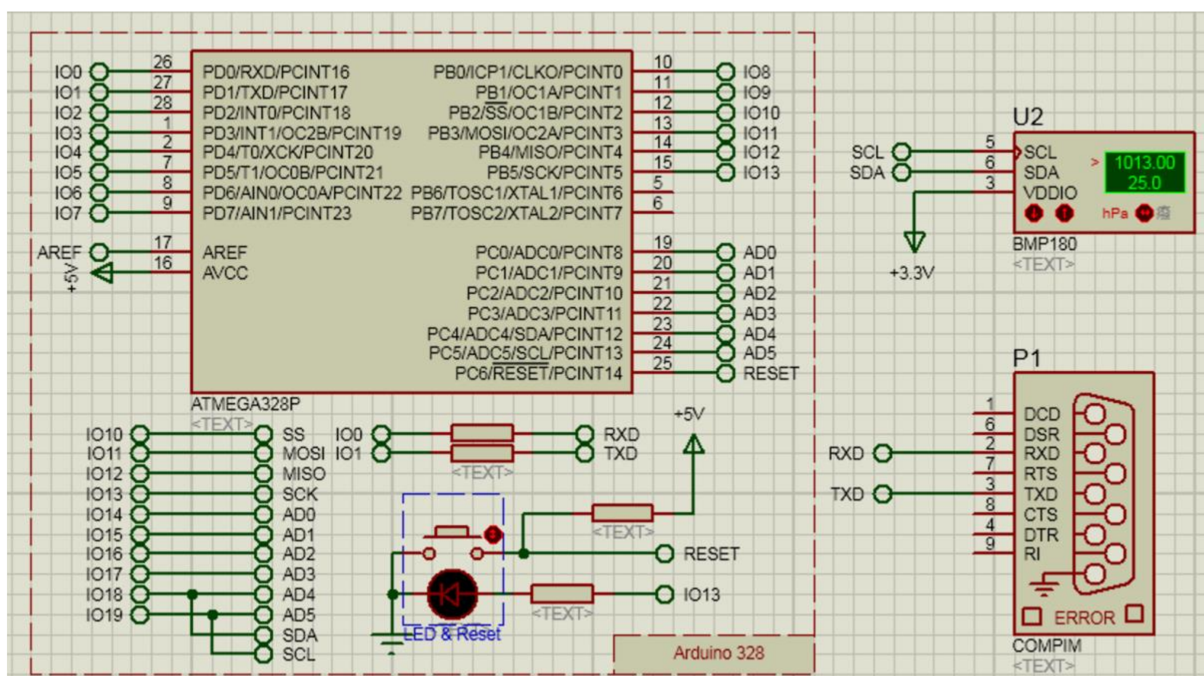
第14章 BMP180 气压传感器实验

14.1 BMP180 气压传感器介绍

BMP180 是一个高精度、超小体积的气压传感器，通过 I2C 进行数据通信。其量程为 300-1100hPa（即 30-110kPa），对应海拔-500m 到+9000 米。

14.2 硬件设计

BMP180 使用 I2C 进行数据通信，故需要将数据线 SDA 和时钟线 SCL 对应接到 Arduino 的 I2C 接口上。如使用多个 I2C 接口的传感器，也可将其接到两个数字 IO 上，并通过自编程实现模拟 I2C，此时不能使用利用 I2C 接口的库。硬件连接如下：



14.3 软件设计

BMP180 和 BMP085 均可以使用 Adafruit BMP085 Library 库，依赖 Adafruit 库。该库使用板载硬件 I2C 接口，将读取 BMP085/BMP180 数据封装成函数。

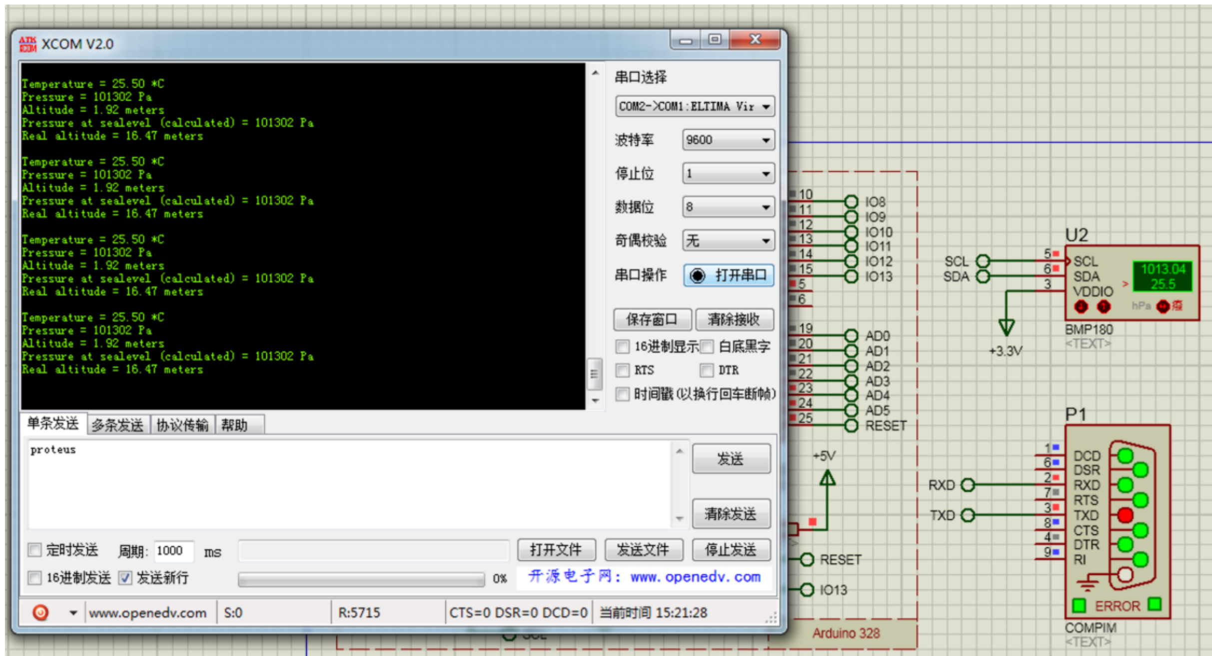
BMP180 气压传感器实验代码如下：

```
#include <Arduino.h>

#include <Wire.h>

#include <Adafruit_BMP085.h>
```

```
Adafruit_BMP085 bmp;
void setup() {
  Serial.begin(9600);
  if (!bmp.begin()) {
   Serial.println("Could not find a valid BMP085 sensor, check wiring!");
   while (1) {}
  }
}
void loop() {
    Serial.print("Temperature = ");
    Serial.print(bmp.readTemperature());
    Serial.println(" *C");
    Serial.print("Pressure = ");
    Serial.print(bmp.readPressure());
    Serial.println(" Pa");
    // 1013.25 millibar = 101325 Pascal
    Serial.print("Altitude = ");
    Serial.print(bmp.readAltitude());
    Serial.println(" meters");
    Serial.print("Pressure at sealevel (calculated) = ");
    Serial.print(bmp.readSealevelPressure());
    Serial.println(" Pa");
    // 通过已知海拔标准气压求真实海拔高度
    Serial.print("Real altitude = ");
    Serial.print(bmp.readAltitude(101500));
    Serial.println(" meters");
    Serial.println();
    delay(500);
}
```

14.4 仿真测试

与其他传感器实验类似，Proteus 中的 BMP180 模块可以调节当前环境的气压和温度。能够看出读取到的气压值还是有一点误差的。



第四篇 FreeRTOS 实时系统篇

基础实验篇和传感器实验篇均为 Arduino 的裸机编程，本篇将在 Arduino 中运行实时操作系统（RTOS）。常用的 RTOS 有国外的 μC/OS、FreeRTOS、RTX 和国内的 RT-Thread、HUAWEI LiteOS、AliOS-Things 等。由于 FreeRTOS 市场占有率高，开源且免费，同时在 Arduino 中有编写好的 FreeRTOS 库，故本篇选用 FreeRTOS 进行讲解。

在裸机系统中，所有的操作都是在一个无限的大循环中实现，但随着要实现的功能越来越多，裸机系统不但不能够很好地解决问题，同时也会使得编程变得更加复杂。RTOS 处于计算机系统的操作系统级，能够更好地进行硬件资源分配，同时其多线程的特性也使得它能够实现比裸机系统更复杂的功能，这也是 RTOS 的最大优势。

本篇分为如下 6 个章节，包含 4 个实验：

第 15 章 FreeRTOS 入门

第 16 章 FreeRTOS 任务基础知识

第 17 章 任务创建与删除实验

第 18 章 中断与二值信号量实验

第 19 章 优先级翻转与互斥信号量实验

第 20 章 消息队列实验

第15章 FreeRTOS 入门

15.1 FreeRTOS 简介

从名字可以看出，FreeRTOS 是一个免费的、自由的 RTOS 类系统。操作系统允许多个任务同时运行，利用任务调度器决定在某一时刻运行某一任务，比如 Unix 系统就是采用时间片轮转的方式进行任务调度。RTOS 的任务调度器被设计为可预测的，这也是实时系统所要求的，为了达到实时系统能够对某一个事件作出实时的响应这一目的。FreeRTOS 是通过任务优先级来决定下一时刻应该运行哪个任务。

FreeRTOS 与 μC/OS 类似，均可在资源受限的微控制器中运行。实际上，从 RTOS 学习的角度看，μC/OS 中文资料多，而 FreeRTOS 大多为英文资料，本课程选用 FreeRTOS 主要从以下几点考虑：

（1）FreeRTOS 商用免费且开源，这是最重要的一点。μC/OS 从某个版本之后不再提供源代码，而是以插件包的形式提供，并且商用需授权；

（2）许多半导体厂商自身产品的 SDK 中就使用 FreeRTOS 作为其操作系统，尤其是关于 Wi-Fi、蓝牙等带有协议栈的模块；

（3）许多软件厂商也使用 FreeRTOS 作为操作系统，比如 ST 公司所有要使用 RTOS 的例程均采用 FreeRTOS；

（4）简单小巧，FreeRTOS 的文件数量少，比 μC/OS-II 和 μC/OS-III 小得多，便于学习和理解；

（5）在 Arduino 中有已经编写好的 FreeRTOS 库，无需自己动手移植，极大的方便了学习和使用。

作为一个可裁剪的小型 RTOS 系统，FreeRTOS 特点包括：

（1）任务调度支持抢占式、合作式和时间片轮转；

（2）系统简单小巧易用，通常情况下内核只占用 4k-9k 字节；

（3）高可移植性，代码主要由 C 语言编写；

（4）支持实时任务和协程（Co-Routines）；

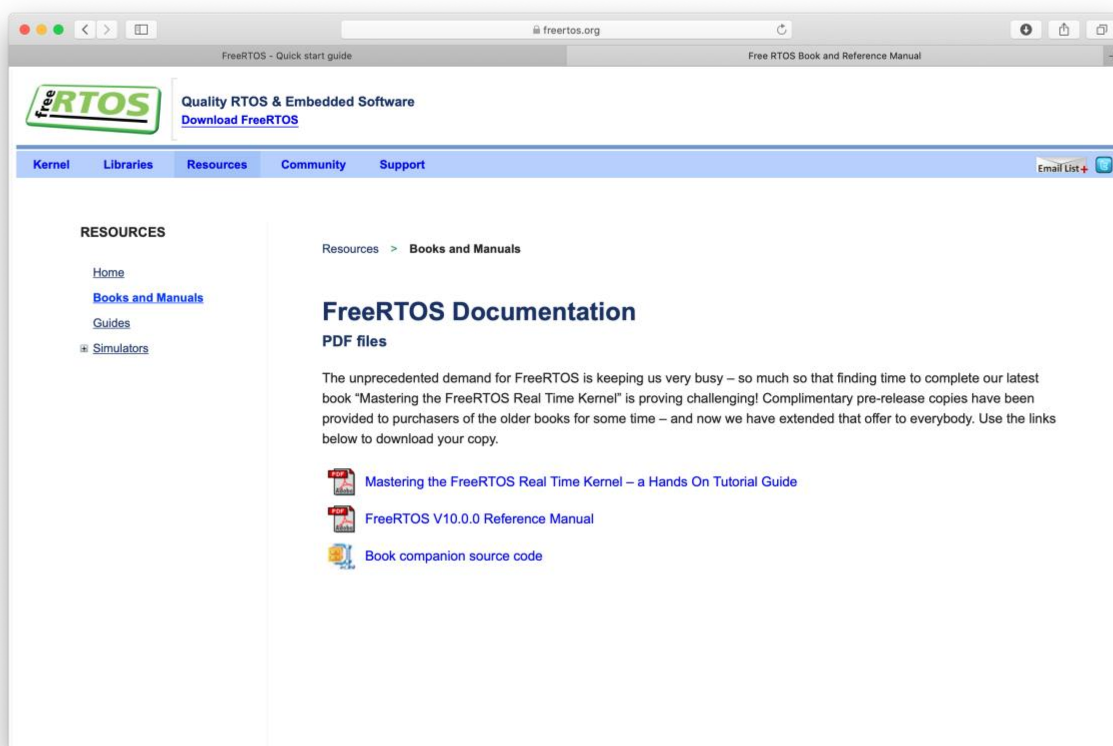
（5）任务和任务、任务与中断之间可以使用任务通知、消息队列、二值信号量、数值型信号量、互斥信号量进行通信和同步；

（6）内置堆栈溢出检测功能；

（7）任务数量及任务优先级不限（μC/OS-II 仅支持 64 个优先级）。

15.2 FreeRTOS 官方文档

FreeRTOS 的文档可以从 FreeRTOS 官方网站 https://www.freertos.org/获取，在 Getting Started 中有很多教程，同时在 Books and Manuals 中有两份 PDF 文档，一份是 FreeRTOS 内核的指导手册，一份是 FreeRTOS 的 API 参考手册。

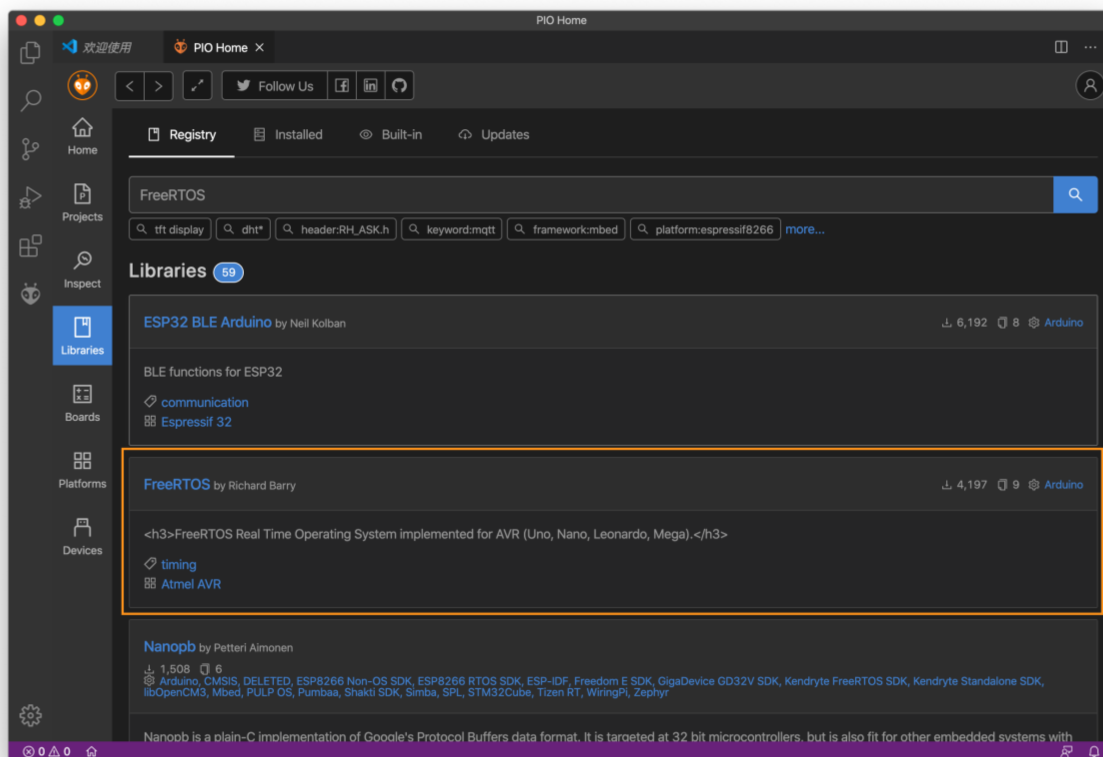


15.3 FreeRTOS 源码获取

FreeRTOS 的源码可以在官方网站直接下载，也可以在 Github 中进行下载。本篇使用 Arduino 的 FreeRTOS 库，可以通过 Arduino IDE 的库管理进行下载，也可以通过 PlatformIO IDE 下载。

通过 Arduino IDE 下载：

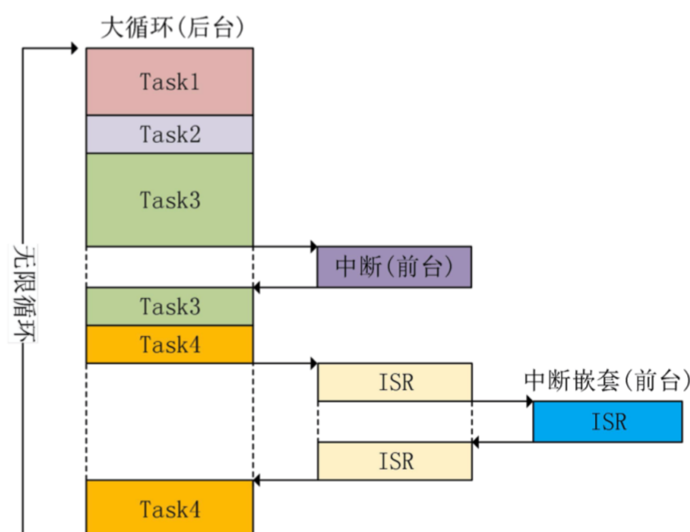
通过 PlatformIO IDE 下载：



第16章 FreeRTOS 任务基础知识

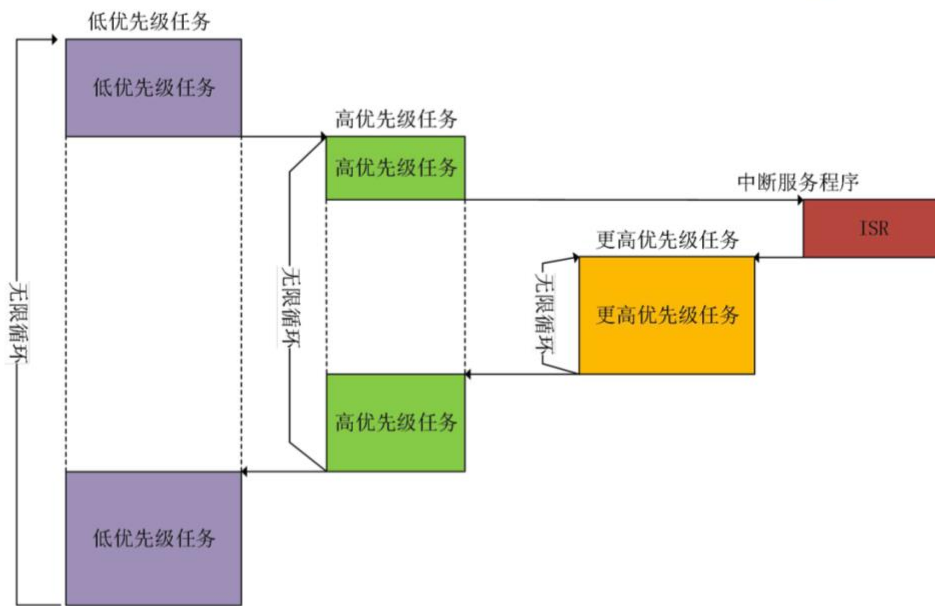
16.1 裸机系统和多任务系统

在基础实验篇和传感器实验篇中，我们在 Arduino 中使用的是裸机系统，在 setup()之后通过 loop()实现一个大循环来完成所有的处理，即应用程序是一个无限的循环，循环中调用相应的函数完成所需的处理，有时候也需要调用中断去完成一些处理。裸机系统又称为单任务系统，也称前后台系统，中断服务函数作为前台程序，大循环 loop()作为后台程序，如图所示：



前后台系统的实时性差，前后台系统各个任务都是排队等着轮流执行，不管这个程序现在有多紧急，没轮到就只能等着，相当于所有任务的优先级都是一样的。前后台系统唯一的优点是系统简单、资源消耗少，但是无法实现复杂功能。

多任务系统会把一个大问题划分成很多个小问题，逐步的把小问题解决掉，大问题也就随之解决了，这些小问题可以单独的作为一个小任务来处理。这些小任务是并发处理的，注意，并不是说同一时刻一起执行很多个任务，而是由于每个任务执行的时间很短，导致看起来像是同一时刻执行了很多个任务一样。多个任务带来了一个新的问题，究竟哪个任务先运行，哪个任务后运行呢？完成这个功能的东西在 RTOS 系统中叫做任务调度器。不同的系统其任务调度器的实现方法也不同，比如 FreeRTOS 是一个抢占式的实时多任务系统，那么其任务调度器也是抢占式的，运行过程如图所示：



高优先级的任务可以打断低优先级任务的运行而取得 CPU 的使用权，这样就保证了那些紧急任务的运行。这样我们就可以为那些对实时性要求高的任务设置一个很高的优先级，比如自动驾驶中的障碍物检测任务等。高优先级的任务执行完成以后重新把 CPU 的使用权归还给低优先级的任务，这个就是抢占式多任务系统的基本原理。

16.2 FreeRTOS 任务与协程

在 FreeRTOS 中应用既可以使用任务，也可以使用协程（Co-Routine），或者两者混合使用。但是任务和协程使用不同的 API 函数，因此不能通过队列或信号量将数据从任务发送给协程，反之亦然。协程是为那些资源很少的 MCU 准备的，其开销很小，但是 FreeRTOS 官方已经不打算再更新协程了，同时 Arduino 的 FreeRTOS 库也没有使用协程，所以在之后的讲解中不再提协程。

在使用 RTOS 的时候一个实时应用可以作为一个独立的任务。每个任务都有自己的运行环境，不依赖于系统中其他的任务或者 RTOS 调度器。任何一个时间点只能有一个任务运行，具体运行哪个任务是由 RTOS 调度器来决定的，RTOS 调度器因此就会重复的开启、关闭每个任务。任务不需要了解 RTOS 调度器的具体行为，RTOS 调度器的职责是确保当一个任务开始执行的时候其上下文环境（寄存器值、堆栈内容等）和任务上一次退出的时候相同。为了做到这一点，每个任务都必须有个堆栈，当任务切换的时候将上下文环境保存在堆栈中，这样当任务再次执行的时候就可以从堆栈中取出上下文环境，任务恢复运行。

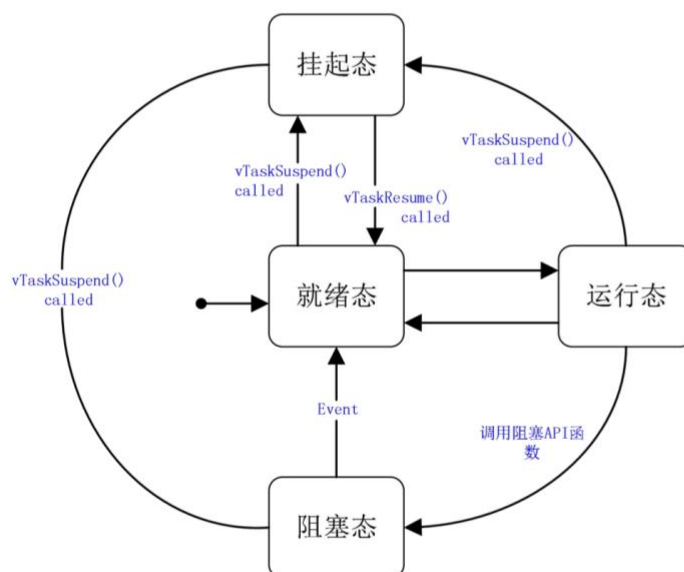
协程是为资源很少的微控制器设计的，与任务相比，所有的协程共用一个堆栈；协程使用合作式的调度器，但是可以在抢占式调度器中使用协程；协程是通过宏定义来实现的，而任务是通过任务函数来实现的；协程为了降低对 RAM 的消耗增加了许多限制。

16.3 FreeRTOS 任务状态

FreeRTOS 中的任务永远处于下面几个状态中的某一个：

- 运行态：当一个任务正在运行时，那么就说这个任务处于运行态，处于运行态的任务就是当前正在使用处理器的任务。如果使用的是单核处理器的话那么不管在任何时刻永远都只有一个任务处于运行态。
- 就绪态：处于就绪态的任务是那些已经准备就绪（这些任务没有被阻塞或者挂起），可以运行的任务，但是处于就绪态的任务还没有运行，因为有一个同优先级或者更高优先级的任务正在运行。
- 阻塞态：如果一个任务当前正在等待某个外部事件的话就说它处于阻塞态，比如说如果某个任务调用了函数 vTaskDelay()的话就会进入阻塞态，直到延时周期完成。任务在等待队列、信号量、事件组、通知或互斥信号量的时候也会进入阻塞态。任务进入阻塞态会有一个超时时间，当超过这个超时时间任务就会退出阻塞态，即使所等待的事件还没有来临。
- 挂起态：像阻塞态一样，任务进入挂起态以后也不能被调度器调用进入运行态，但是进入挂起态的任务没有超时时间。任务进入和退出挂起态通过调用函数 vTaskSuspend()和 xTaskResume()实现。

任务状态转换如图所示：



16.4 FreeRTOS 任务优先级

每个任务都可以分配一个从 0 到 configMAX_PRIORITIES - 1 的优先级，configMAX_PRIORITIES 在文件 FreeRTOSConfig.h 中有定义。这个宏通常可以为任意值，但是考虑到 RAM 的消耗，最好设置为一个满足应用的最小值。FreeRTOS 中优先级数字越低表示任务的优先级越低，0 的优先级最低，configMAX_PRIORITIES-1 的优先级最高。空闲任务的优先级最低，为 0。

FreeRTOS 调度器确保处于就绪态或运行态的高优先级的任务获取处理器使用权，换句话说就是处于就绪态的最高优先级的任务才会运行。当宏 configUSE_TIME_SLICING 定义为 1 的时候多个任务可以共用一个优先级，数量不限，此时处于就绪态的优先级相同的任务就会使用时间片轮转调度器获取运行时间。然而，Arduino 的 FreeRTOS 库未实现时间片调度，除非任务使用 vTaskDelay()等内置任务切换的函数，否则不会触发任务调度。

16.5 FreeRTOS 任务实现

在使用 FreeRTOS 的过程中，要使用函数 xTaskCreate()，这个函数的第一个参数 pxTaskCode 就是这个任务的任务函数。任务函数就是完成本任务工作的函数。这个任务要完成什么样的功能都是在这个任务函数中实现的。比如这个任务要点个流水灯，那么这个流水灯的代码就是任务函数中实现的。FreeRTOS 官方给出的任务函数模板如下：

```
void vATaskFunction( void *pvParameters ){
    for( ;; ){
        -- 任务应用程序代码 --
    }
    /* 任务不能从任务函数中返回或退出。只能通过调用函数将任务删除 */
    vTaskDelete( NULL );
}
```

（1）任务函数本质是函数，函数名就是任务名，但是任务函数必须没有返回值（即返回值类型 void）；

（2）任务的具体执行过程为大循环，类似于一个单任务系统；

（3）FreeRTOS 任务中延时应该使用 vTaskDelay()，才能保证在任务被阻塞时也能按照要求精确延时；

（4）任务一般不允许跳出循环，如果一定要停止此任务需要用任务删除函数，等需要时再重新创建任务。

16.6 FreeRTOS 任务控制块

FreeRTOS 的每个任务都有一些属性需要存储，FreeRTOS 把这些属性集合到一起用一个结构体来表示，这个结构体叫做任务控制块 TCB_t。在使用函数 xTaskCreate()创建任务的时候就会自动的给每个任务分配一个任务控制块。TCB_t 结构体在文件 tasks.c 中有定义，如下：

```
typedef struct TaskControlBlock_t {
    volatile StackType_t    *pxTopOfStack;   // 任务堆栈栈顶
    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS    xMPUSettings;        // 与 MPU 有关
    #endif
    ListItem_t          xStateListItem;      // 状态列表项
    ListItem_t          xEventListItem;      // 事件列表项
    UBaseType_t          uxPriority;          // 任务优先级
    StackType_t         *pxStack;             // 任务堆栈起始地址
    char                pcTaskName[ configMAX_TASK_NAME_LEN ];   // 任务名字
    #if (( portSTACK_GROWTH > 0 )||( configRECORD_STACK_HIGH_ADDRESS == 1 ))
        StackType_t      *pxEndOfStack;       // 任务堆栈栈底
    #endif
    #if ( portCRITICAL_NESTING_IN_TCB == 1 )
        UBaseType_t      uxCriticalNesting;   // 临界区嵌套深度
    #endif
    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t      uxTCBNumber;         // DEBUG 时使用
        UBaseType_t      uxTaskNumber;        // DEBUG 时使用
    #endif
    #if ( configUSE_MUTEXES == 1 )
        UBaseType_t      uxBasePriority;      // 任务基础优先级，优先级反转时使用
        UBaseType_t      uxMutexesHeld;       // 任务获取到的互斥信号量个数
    #endif
    #if ( configUSE_APPLICATION_TASK_TAG == 1 )
        TaskHookFunction_t pxTaskTag;
    #endif
    #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )   // 与本地存储有关
```

```
        void
*pvThreadLocalStoragePointers[ configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
    #endif
    #if( configGENERATE_RUN_TIME_STATS == 1 )
        uint32_t        ulRunTimeCounter;    // 记录任务运行总时间
    #endif
    #if ( configUSE_NEWLIB_REENTRANT == 1 )
        struct    _reent xNewLib_reent;
    #endif
    #if( configUSE_TASK_NOTIFICATIONS == 1 )
        volatile uint32_t ulNotifiedValue;    // 任务通知值
        volatile uint8_t ucNotifyState;       // 任务通知状态
    #endif
    #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
        uint8_t    ucStaticallyAllocated;    // 标记任务是动态创建的还是静态创建的
    #endif
    #if( INCLUDE_xTaskAbortDelay == 1 )
        uint8_t ucDelayAborted;
    #endif
    #if( configUSE_POSIX_ERRNO == 1 )
        int iTaskErrno;
    #endif
} TCB_t;
```

可以看出 FreeRTOS 的任务控制块中的成员变量相比 μC/OS-III 要少很多，而且大多数与裁剪有关，当不使用某些功能的时候与其相关的变量就不参与编译，任务控制块大小就会进一步的减小。

16.7 FreeRTOS 任务堆栈

FreeRTOS 之所以能正确的恢复一个任务的运行，就是因为有任务堆栈在保驾护航，任务调度器在进行任务切换的时候会将当前任务的现场（CPU 寄存器值等）保存在此任务的任务堆栈中，等到此任务下次运行的时候就会先用堆栈中保存的值来恢复现场，恢复现场以后任务就会接着从上次中断的地方开始运行。

创建任务的时候需要给任务指定堆栈，如果使用的函数 xTaskCreate()动

态创建任务的话那么任务堆栈就会自动创建，后面分析 xTaskCreate()的时候会讲解。

不管是使用函数 xTaskCreate()还是 xTaskCreateStatic()创建任务都需要指定任务堆栈大小。任务堆栈的数据类型为 StackType_t，StackType_t 本质上是 uint32_t，在 portmacro.h 中有定义，StackType_t 类型的变量为 4 个字节，那么任务的实际堆栈大小就应该是所定义的 4 倍。

第17章 任务创建与删除实验

17.1 任务创建和删除 API 函数

FreeRTOS 最基本的功能就是任务管理，而任务管理最基本的操作就是创建和删除任务。任务的创建 API 函数有两个，xTaskCreate()函数用于动态创建任务，而 xTaskCreateStatic()函数用于静态创建任务，在本书实验中只考虑动态创建任务。

1、xTaskCreate()函数

此函数用来创建一个任务，任务需要 RAM 来保存与任务有关的状态信息（任务控制块），任务也需要一定的 RAM 来作为任务堆栈。如果使用函数 xTaskCreate()来创建任务的话，这些所需的 RAM 就会自动的从 FreeRTOS 的堆中分配，因此必须提供内存管理文件，Arduino 的 FreeRTOS 库使用 heap_3.c 这个内存管理文件。新创建的任务默认就是就绪态的，如果当前没有比它更高优先级的任务运行，那么此任务就会立即进入运行态开始运行，不管在任务调度器启动前还是启动后，都可以创建任务。函数原型如下：

```
BaseType_t xTaskCreate(
    TaskFunction_t            pvTaskCode,     // 任务函数
    const char * const        pcName,         // 任务名字
    configSTACK_DEPTH_TYPE    usStackDepth,   // 任务堆栈大小
    void *                    pvParameters,   // 传递给任务的参数
    UBaseType_t               uxPriority,     // 任务优先级
    TaskHandle_t *            pvCreatedTask);// 任务句柄
```

任务创建成功返回 pdPASS，任务创建失败返回对应的 error code。

2、vTaskDelete()函数

此函数用于删除一个由 xTaskCreate()创建的任务，被删除了的任务不再存在，也就是说再也不会进入运行态。任务被删除以后就不能再使用此任务的句柄。如果此任务是使用动态方法创建的，那么在此任务被删除以后，此任务之前申请的堆栈和控制块内存会在空闲任务中被释放掉，因此当调用函数 vTaskDelete()删除任务以后必须给空闲任务一定的运行时间。

只有那些由内核分配给任务的内存才会在任务被删除以后自动的释放掉，用户分配给任务的内存需要用户自行释放掉，比如某个任务中用户调用函数 pvPortMalloc()分配了 500 字节的内存，那么在此任务被删除以后用户也必须调用函数 vPortFree()将这 500 字节的内存释放掉，否则会导致内存

泄露。vTaskDelete()函数原型如下：

```
void vTaskDelete( TaskHandle_t xTask );  // 要删除的任务句柄
```

此函数没有返回值。

17.2 实验设计

17.2.1 实验内容设计

本实验设计 3 个任务：start_task、led0_task 和 led1_task，这三个任务的功能如下：

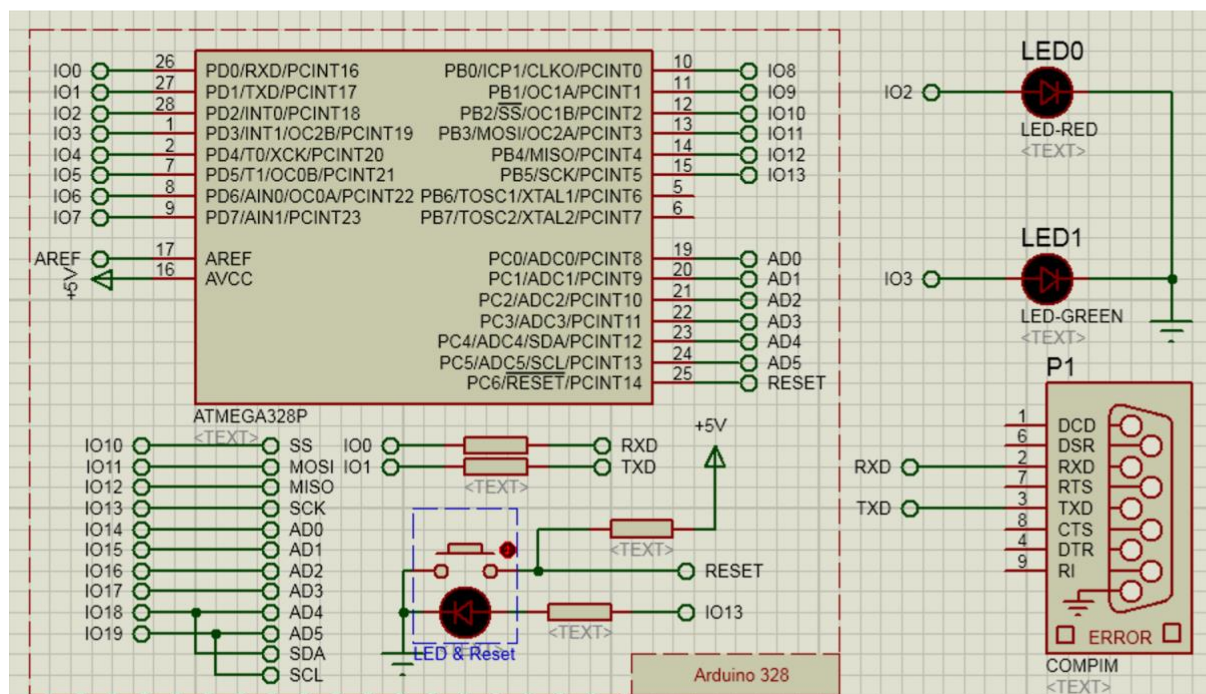
start_task 用于创建 led0_task 和 led1_task；

led0_task 用于控制 LED0 闪烁，通过串口输出任务运行次数，同时当任务运行 5 次之后，将 led1_task 删除；

led1_task 用于控制 LED1 闪烁，通过串口输出任务运行次数。

17.2.2 硬件设计

为了增强实验的显示效果，不使用板载的黄色 LED 灯，用红色 LED 表示 LED0，用绿色 LED 表示 LED1，硬件连接如下：



17.2.3 软件设计

start_task 任务用于创建 led0_task 和 led1_task 两个任务，创建后将 start_task 任务本身删除。start_task 在创建两个任务时进入了临界区，为了

保证创建完两个任务后才执行其他任务。任务创建与删除实验代码如下：

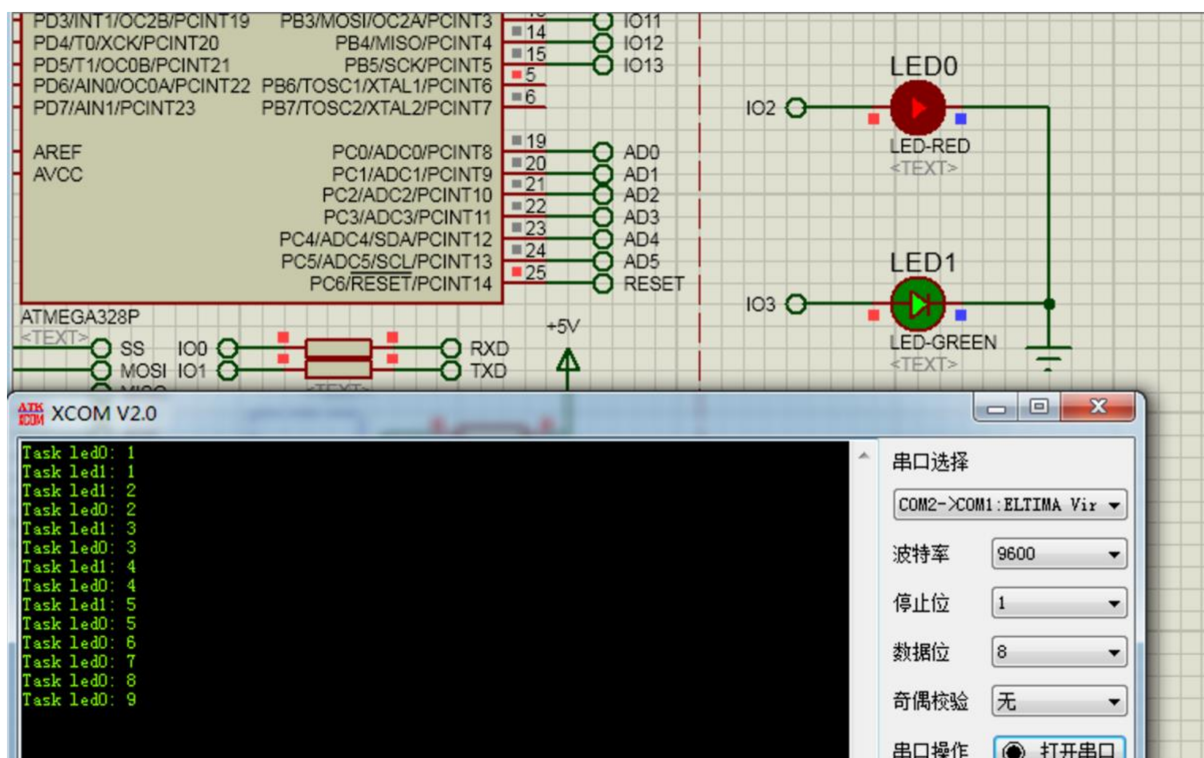
```
#include <Arduino.h>
#include <Arduino_FreeRTOS.h>
int LED0_PIN = 2;
int LED1_PIN = 3;
TaskHandle_t StartTask_Handler;
TaskHandle_t LED0Task_Handler;
TaskHandle_t LED1Task_Handler;
void start_task(void* pvParameters);
void led0_task(void* pvParameters);
void led1_task(void* pvParameters);
void setup() {
  pinMode(LED0_PIN, OUTPUT);
  pinMode(LED1_PIN, OUTPUT);
  digitalWrite(LED0_PIN, LOW);
  digitalWrite(LED1_PIN, LOW);
  Serial.begin(9600);
  xTaskCreate(              // 创建开始任务
    start_task,
    "start_task",
    128,
    NULL,
    1,
    &StartTask_Handler);
  vTaskStartScheduler();    // 开启任务调度
}
void loop() {
  // 空，程序放在任务中执行。
}
void start_task(void* pvParameters) {
  taskENTER_CRITICAL();     // 进入临界区
  xTaskCreate(              // 创建 LED0 任务
    led0_task,
    "led0_task",
```

```
        128,
        NULL,
        2,
        &LED0Task_Handler);
    xTaskCreate(                // 创建 LED1 任务
        led1_task,
        "led1_task",
        128,
        NULL,
        3,
        &LED1Task_Handler);
    vTaskDelete(StartTask_Handler);   // 删除 start_task
    taskEXIT_CRITICAL();      // 退出临界区
}
void led0_task(void* pvParameters) {
    int led0_num = 0;
    for(;;) {
        led0_num++;
        Serial.print("Task led0: ");
        Serial.println(led0_num);
        if(led0_num == 5) {
            vTaskDelete(LED1Task_Handler);   // 删除 LED1 任务
        }
        digitalWrite(LED0_PIN, HIGH);
        vTaskDelay(50);
        digitalWrite(LED0_PIN, LOW);
        vTaskDelay(50);
    }
}
void led1_task(void* pvParameters) {
    int led1_num = 0;
    for(;;) {
        led1_num++;
        Serial.print("Task led1: ");
```

```
        Serial.println(led1_num);
        digitalWrite(LED1_PIN, HIGH);
        vTaskDelay(50);
        digitalWrite(LED1_PIN, LOW);
        vTaskDelay(50);
    }
}
```

17.3 仿真测试

运行 Proteus 和串口调试助手，刚开始红绿两个 LED 均闪烁，当任务 led0_task 运行 5 次以后，led1_task 任务被删除，串口不再输出其运行次数，同时绿灯不再闪烁，保持常亮（或常灭）。



第18章 中断与二值信号量实验

18.1 二值信号量简介

信号量是操作系统中重要的一部分，信号量一般用来进行资源管理和任务同步，FreeRTOS 中信号量又分为二值信号量、计数型信号量、互斥信号量和递归互斥信号量。不同的信号量其应用场景不同，但有些应用场景是可以互换着使用的。

二值信号量通常用于互斥访问或同步，二值信号量和互斥信号量非常类似，但是还是有一些细微的差别，互斥信号量拥有优先级继承机制，二值信号量没有优先级继承。因此二值信号量更适合用于同步（任务与任务或任务与中断的同步），而互斥信号量适合用于简单的互斥访问。本节只讲解二值信号量在同步中的应用。

和消息队列（第 20 章讲述）一样，信号量 API 函数允许设置一个阻塞时间，阻塞时间是当任务获取信号量的时候，由于信号量无效，从而导致任务进入阻塞态的最大时钟节拍数。如果多个任务同时阻塞在同一个信号量上，那么优先级最高的任务优先获得信号量，这样当信号量有效的时候高优先级的任务就会解除阻塞状态。

二值信号量其实就是一个只有一个队列项的队列，这个特殊的队列要么是满的，要么是空的，就是所谓的二值。任务和中断使用这个特殊队列不用在乎队列中存的是什么消息，只需要知道这个队列是满的还是空的。可以利用这个机制来完成任务与中断之间的同步。

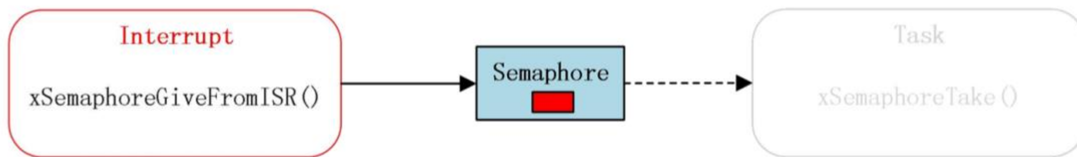
在实际应用中通常会使用一个任务来处理 MCU 的某个外设，比如网络应用中，一般最简单的方法就是使用一个任务去轮询的查询 MCU 的 ETH 外设是否有数据，当有数据的时候就处理这个网络数据。这样使用轮询的方式是很浪费 CPU 资源的，而且也阻止了其他任务的运行。最理想的方法就是当没有网络数据的时候网络任务就进入阻塞态，把 CPU 让给其他的任务，当有数据的时候网络任务才去执行。现在使用二值信号量就可以实现这样的功能，任务通过获取信号量来判断是否有网络数据，没有的话就进入阻塞态，而网络中断服务函数通过释放信号量来通知任务以太网外设接收到了网络数据，网络任务可以去提取处理了。网络任务只是在一直的获取二值信号量，它不会释放信号量，而中断服务函数是一直在释放信号量，它不会获取信号量。也可以使用任务通知功能来替代二值信号量，而且使用任务通知的话速度更快，代码量更少，有关任务通知的内容可以自行学习。

使用二值信号量来完成中断与任务同步的这个机制中，任务优先级确保了外设能够得到及时的处理，这样做相当于推迟了中断处理过程。也可以使用消息队列来替代二值信号量，在外设事件的中断服务函数中获取相关数据，并将相关的数据通过消息队列发送给任务。如果消息队列无效的话任务就进入阻塞态，直至队列中有数据，任务接收到数据以后就开始相关的处理过程，消息队列将在第 20 章讲述。FreeRTOS 在线文档中有关于二值信号量工作过程的动态图讲解，分解如下：

1、二值信号量无效

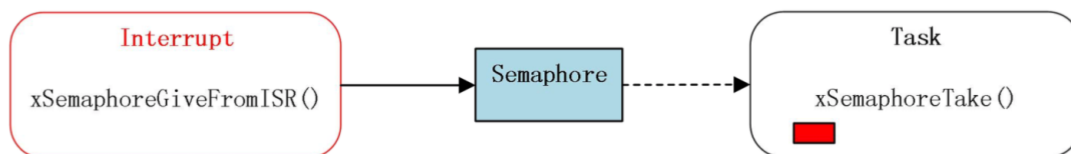
任务 Task 通过函数 xSemaphoreTask()获取信号量，但此时二值信号量无效，所以任务 Task 进入阻塞态。

2、中断释放信号量



此时中断发生，在中断服务函数中通过函数 xSemaphoreGiveFromISR()释放信号量，因此信号量变为有效。

3、任务获取信号量成功



由于信号量已经有效了，所以任务 Task 获取信号量成功，任务从阻塞态解除，开始执行相关的处理过程。

4、任务再次进入阻塞态

由于任务函数一般都是一个大循环，所以在任务做完相关的处理以后就会再次调用函数 xSemaphoreTake()获取信号量。在执行完第三步以后二值信号量就已经变为无效的了，所以任务将再次进入阻塞态，和第一步一样，直至中断再次发生并且调用函数 xSemaphoreGiveFromISR()释放信号量。

18.2 创建二值信号量

18.2.1 创建二值信号量 API 函数

要想使用二值信号量，就必须先创建一个二值信号量。同任务创建类似，二值信号量的创建同样分为动态方法和静态方法，同样的，静态方法创建信号量所需的 RAM 空间由用户分配，本章只考虑动态创建二值信号量。

函数 xSemaphoreCreateBinary()

此函数是新版本 FreeRTOS 创建二值信号量的函数。使用此函数创建二值信号量的话信号量所需要的 RAM 是由 FreeRTOS 的内存管理部分来动态分配的。此函数创建好的二值信号量默认是空的，也就是说刚创建好的二值信号量使用函数 xSemaphoreTake()是获取不到的，此函数也是个宏，具体创建过程是由函数 xQueueGenericCreate()来完成的，函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateBinary( void )
```

创建成功函数返回创建的二值信号量的句柄，失败返回 NULL。

18.2.2 二值信号量创建过程分析

分析二值信号量创建过程，函数 xSemaphoreCreateBinary()定义如下：

```
#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define xSemaphoreCreateBinary()                    \
            xQueueGenericCreate( ( UBaseType_t ) 1,     \
            semSEMAPHORE_QUEUE_ITEM_LENGTH,             \
            queueQUEUE_TYPE_BINARY_SEMAPHORE )
#endif
```

二值信号量是在队列的基础上实现的，所以创建二值信号量就是创建队列的过程。这里使用函数 xQueueGenericCreate()创建了一个队列，队列长度为 1，队列项长度为 0，队列类型为二值信号量。

新版本创建二值信号量后没有立刻调用 xSemaphoreGive()释放信号量，也就是说，新版本函数创建的二值信号量默认是无效的。

18.3 释放二值信号量

同消息队列一样，释放信号量也分为任务级和中断级。此外，除了递归互斥信号量有专用的释放函数，其他信号量，包括二值信号量、计数型信号量和互斥信号量，均使用同样的函数释放信号量。任务级释放函数为 xSemaphoreGive()，中断级释放函数为 xSemaphoreGiveFromISR()。

1、函数 xSemaphoreGive()

此函数用于释放二值信号量、计数型信号量或互斥信号量，此函数是一个宏，真正释放信号量的过程是由函数 xQueueGenericSend()来完成的，函数原型如下：

```
BaseType_t xSemaphoreGive( xSemaphore )  // 要释放的信号量句柄
```

释放成功返回 pdPASS，失败返回对应的 error code。

再看一下 xSemaphoreGive()的具体内容：

```
#define xSemaphoreGive( xSemaphore )                              \
        xQueueGenericSend( ( QueueHandle_t ) ( xSemaphore ),   \
        NULL,                                                     \
        semGIVE_BLOCK_TIME,                                       \
        queueSEND_TO_BACK )
```

可以看出任务级释放信号量就是向队列发送消息的过程，只是这里并没有发送具体的消息，阻塞时间为 0，入队方式采用的后向入队。也能够看出，二值信号量就是通过队列的满与空来确定是否有效。

2、函数 xSemaphoreGiveFromISR()

此函数用于在中断中释放信号量，此函数只能用来释放二值信号量和计数型信号量，绝对不能用来在中断服务函数中释放互斥信号量。此函数是一个宏，真正执行的是函数 xQueueGiveFromISR()，此函数原型如下：

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,
                                  BaseType_t*  pxHigherPriorityTaskWoken)
```

参数 xSemaphore 为要释放的信号量句柄，pxHigherPriorityTaskWoken 用来标记退出此函数以后是否进行任务切换，这个变量的值用户不用进行设置，只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。释放信号量成功返回 pdPASS，失败返回对应的 error code。

18.4 获取二值信号量

获取信号量也分为任务级和中断级，同释放信号量的 API 函数一样，不管是二值信号量、计数型信号量还是互斥信号量，均使用同一个获取函数。

1、函数 xSemaphoreTake()

此函数用于获取二值信号量、计数型信号量或互斥信号量，此函数是一个宏，真正获取信号量的过程是由函数 xQueueGenericReceive()来完成的，函数原型如下：

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore,// 信号量句柄
                          TickType_t        xBlockTime)// 阻塞时间
```

获取信号量成功返回 pdTRUE，超时或获取失败返回 pdFALSE。

再看一下 xSemaphoreTake()的具体内容：

```
#define xSemaphoreTake( xSemaphore, xBlockTime )                    \
        xQueueGenericReceive( ( QueueHandle_t ) ( xSemaphore ), \
        NULL,                                                   \
        ( xBlockTime ),                                         \
        pdFALSE )
```

获取信号量的过程其实就是读取队列的过程，只是这里并不是为了读取队列中的消息。xQueueGenericReceive()函数将在第 20 章进行讲解。

2、函数 xSemaphoreTakeFromISR()

此函数用于在中断服务函数中获取信号量，此函数用于获取二值信号量和计数型信号量，绝对不能使用此函数来获取互斥信号量.此函数是一个宏，真正执行的是函数 xQueueReceiveFromISR()，此函数原型如下：

```
BaseType_t xSemaphoreTakeFromISR(SemaphoreHandle_t xSemaphore,
                                 BaseType_t * pxHigherPriorityTaskWoken)
```

参数 xSemaphore 为要获取的信号量句柄，pxHigherPriorityTaskWoken 用来标记退出此函数以后是否进行任务切换，这个变量的值用户不用进行设置，只需要提供一个变量来保存这个值就行了。当此值为 pdTRUE 的时候在退出中断服务函数之前一定要进行一次任务切换。释放信号量成功返回 pdPASS，失败返回 pdFALSE。

18.5 实验设计

18.5.1 实验内容设计

本实验仅设计一个任务，TaskLed，从中断服务函数中获取二值信号量，并改变 LED 状态。

18.5.2 硬件设计

本实验使用外部中断 INT0，硬件连接与第 8 章外部中断实验相同。

18.5.3 软件设计

中断服务函数仅用于释放二值信号量，中断触发为低电平触发，同时配

置外部中断接口时，通过 INPUT_PULLUP 模式使用了 Arduino 内置的上拉电阻，这样在 Proteus 仿真时才能设置为低电平触发。中断与二值信号量实验代码如下：

```
#include <Arduino.h>
#include <Arduino_FreeRTOS.h>
#include <semphr.h>
SemaphoreHandle_t interruptSemaphore; // 声明一个二值信号量
void TaskLed(void *pvParameters);
void interruptHandler();
void setup() {
  pinMode(2, INPUT_PULLUP);
  xTaskCreate(TaskLed,
              "Led",
              128,
              NULL,
              0,
              NULL );
  interruptSemaphore = xSemaphoreCreateBinary();
  if (interruptSemaphore != NULL) {
    // 配置中断
    attachInterrupt(digitalPinToInterrupt(2), interruptHandler, LOW);
  }
}
void loop() {}
void interruptHandler() {
  // 释放信号量
  xSemaphoreGiveFromISR(interruptSemaphore, NULL);
}
void TaskLed(void *pvParameters) {
  (void) pvParameters;
  pinMode(LED_BUILTIN, OUTPUT);
  for (;;) {
    // 获取信号量
    if (xSemaphoreTake(interruptSemaphore, portMAX_DELAY) == pdPASS) {
```

```
            digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
          }
        }
      }
```

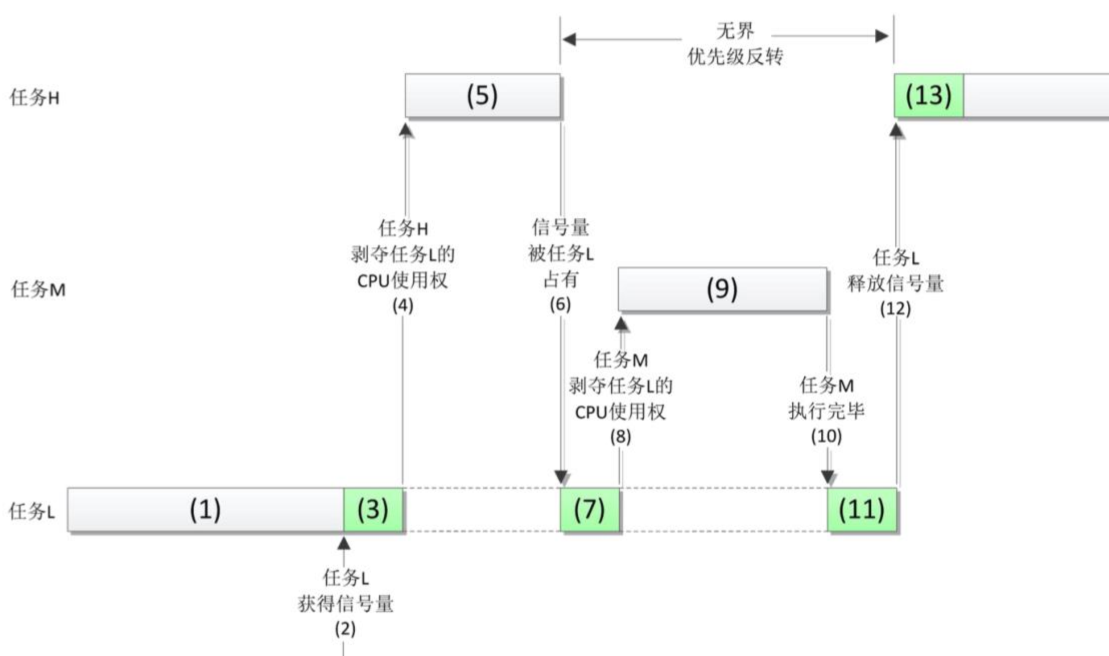
18.6 仿真测试

加载程序文件，运行仿真。TaskLed 任务由于等待信号量而被阻塞，LED 灯状态不发生变化；当按下按键触发中断后，中断服务函数释放信号量，TaskLed 获得信号量进入运行态，LED 状态改变一次，再阻塞等待信号量。

第19章 优先级翻转与互斥信号量实验

19.1 优先级翻转

在使用二值信号量的时候会遇到很常见的一个问题——优先级翻转，优先级翻转在可剥夺内核中是非常常见的，在实时系统中不允许出现这种现象，这样会破坏任务的预期顺序，可能会导致严重的后果。下图是一个优先级翻转的例子：



（1）任务 H 和任务 M 处于挂起状态，等待某一事件的发生，任务 L 正在运行；

（2）某一时刻任务 L 想要访问共享资源，在此之前它必须先获得对应该资源的信号量；

（3）任务 L 获得信号量并开始使用该共享资源；

（4）由于任务 H 优先级高，它等待事件发生后便剥夺了任务 L 的 CPU 使用权；

（5）任务 H 开始运行；

（6）任务 H 运行过程中也要使用任务 L 正在使用着的资源，由于该资源的信号量还被任务 L 占用着，任务 H 只能进入挂起状态，等待任务 L 释放该信号量；

（7）任务 L 继续运行；

（8）由于任务 M 的优先级高于任务 L，当任务 M 等待的事件发生后，

任务 M 剥夺了任务 L 的 CPU 使用权；

（9）任务 M 处理该处理的事；

（10）任务 M 执行完毕后，将 CPU 使用权归还给任务 L；

（11）任务 L 继续运行；

（12）最终任务 L 完成所有的工作并释放了信号量，到此为止，由于实时内核知道有个高优先级的任务在等待这个信号量，故内核做任务切换；

（13）任务 H 得到该信号量并接着运行。

在这种情况下，任务 H 的优先级实际上降到了任务 L 的优先级水平，因为任务 H 要一直等待直到任务 L 释放其占用的共享资源。由于任务 M 剥夺了任务 L 的 CPU 使用权，使得任务 H 的情况更加恶化，这样就相当于任务 M 的优先级高于任务 H，导致优先级翻转。

关于优先级翻转可以做个小实验，实验创建三个任务，有三个不同优先级，对应例子中的任务 L、任务 M 和任务 H。三个任务之间通过二值信号量作互斥，用串口输出任务状态，观察优先级翻转现象。

19.2 互斥信号量简介

互斥信号量其实就是一个拥有优先级继承的二值信号量，在同步的应用中二值信号量最适合，而在需要互斥访问的应用中应该使用互斥信号量。在互斥访问中互斥信号量相当于一个钥匙，当任务想要使用资源的时候就必须先获得这个钥匙，当使用完资源以后就必须归还这个钥匙，这样其他的任务就可以拿着这个钥匙去使用资源。

互斥信号量使用和二值信号量相同的 API 操作函数，所以互斥信号量也可以设置阻塞时间，不同于二值信号量的是，互斥信号量具有优先级继承的特性。当一个互斥信号量正在被一个低优先级的任务使用，而此时有个高优先级的任务也尝试获取这个互斥信号量的话就会被阻塞。不过这个高优先级的任务会将低优先级任务的优先级提升到与自己相同的优先级，这个过程就是优先级继承。优先级继承尽可能的降低了高优先级任务处于阻塞态的时间，并且将已经出现的“优先级翻转”的影响降到最低。

优先级继承并不能完全的消除优先级翻转，它只是尽可能的降低优先级翻转带来的影响。硬实时应用应该在设计之初就要避免优先级翻转的发生。互斥信号量不能用于中断服务函数中，原因如下：

- 互斥信号量有优先级继承的机制，所以只能用在任务中，不能用于中断服务函数；
- 中断服务函数中不能因等待互斥信号量而设置阻塞时间进入阻塞态。

19.3 创建互斥信号量

19.3.1 创建互斥信号量 API 函数

互斥信号量的创建同样分为动态方法和静态方法，静态方法创建信号量所需的 RAM 空间由用户分配，本章只考虑动态创建互斥信号量。

函数 xSemaphoreCreateMutex()

此函数用于创建一个互斥信号量，所需要的内存通过动态内存管理方法分配。此函数本质是一个宏，真正完成信号量创建的是函数 xQueueCreateMutex()，此函数原型如下：

```
SemaphoreHandle_t xSemaphoreCreateMutex( void )
```

创建成功函数返回创建的互斥信号量的句柄，失败返回 NULL。

19.3.2 互斥信号量创建过程分析

创建互斥信号量函数本质上是一个宏，真正完成创建互斥信号量工作的是函数 xQueueCreateMutex()，此函数的定义如下：

```
QueueHandle_t xQueueCreateMutex( const uint8_t ucQueueType ) {
  QueueHandle_t xNewQueue;
  const UBaseType_t uxMutexLength=(UBaseType_t)1,uxMutexSize=(UBaseType_t)0;
  xNewQueue = xQueueGenericCreate( uxMutexLength, uxMutexSize, ucQueueType );
  prvInitialiseMutex(( Queue_t * ) xNewQueue);
  return xNewQueue;
}
```

可以看出，此函数调用函数 xQueueGenericCreate()创建一个队列，队列长度为 1，队列项长度为 0，队列类型为参数 ucQueueType，参数 ucQueueType 为 queueQUEUE_TYPE_MUTEX。

互斥信号量初始化函数 prvInitialiseMutex()中，对此队列某些成员重新赋值，实际上是将队列头和队列尾进行重命名，重命名后一个指向保存互斥队列的所有者，另一个指向拥有互斥信号量的任务的任务控制块。重命名主要是为了增强代码的可读性。最后，互斥信号量初始化函数会调用函数释放一次互斥信号量，即互斥信号量创建后默认就是有效的。

19.4 释放互斥信号量

释放互斥信号量的时候和二值信号量、计数型信号量一样，都是用的函数 xSemaphoreGive()。不过由于互斥信号量涉及到优先级继承的问题，所以

具体处理过程会有点区别。使用函数 xSemaphoreGive()释放信号量最重要的一步就是将 uxMessagesWaiting 加 1，而这一步就是通过函数 prvCopyDataToQueue()来完成的，释放信号量的函数 xQueueGenericSend()会调用 prvCopyDataToQueue()函数。互斥信号量的优先级继承也是在函数 prvCopyDataToQueue()中完成的，其中有这样一段代码：

```
if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX ) {
    xReturn = xTaskPriorityDisinherit(( void * ) pxQueue->pxMutexHolder);
    pxQueue->pxMutexHolder = NULL;
}
```

其中，函数 xTaskPriorityDisinherit()处理互斥信号量的优先级继承问题，具体处理过程可以查看 FreeRTOS 源代码。

19.5 获取互斥信号量

获取互斥信号量的函数同获取二值信号量和计数型信号量的函数相同，都是 xSemaphoreTake()，获取互斥信号量的过程也需要处理优先级继承的问题，函数 xSemaphoreTake()的原型 xQueueGenericReceive()在文件 queue.c 中有定义，此处不详细讲解。

xSemaphoreTake()会调用函数 vTaskPriorityInherit()处理互斥信号量中的优先级继承问题，如果函数 xQueueGenericReceive()用于获取互斥信号量的话，此函数执行到 vTaskPriorityInherit()说明互斥信号量正在被其他的任务占用。vTaskPriorityInherit()和函数 xTaskPriorityDisinherit()过程相反，此函数会判断当前任务的任务优先级是否比正在拥有互斥信号量的那个任务的任务优先级高，如果是的话就会把拥有互斥信号量的那个低优先级任务的优先级调整为与当前任务相同的优先级。

举个例子来简单的演示一下这个过程，假设现在有两个任务 HighTask 和 LowTask，HighTask 的任务优先级为 4，LowTask 的任务优先级为 2。这两个任务都会操同一个互斥信号量 Mutex，LowTask 先获取到互斥信号量 Mutex。此时任务 HighTask 也要获取互斥信号量 Mutex，任务 HighTask 调用函数 xSemaphoreTake()尝试获取互斥信号量 Mutex，发现此互斥信号量正在被任务 LowTask 使用，并且 LowTask 的任务优先级为 2，比自己的任务优先级小，任务 HighTask 就会将 LowTask 的任务优先级调整为与自己相同的优先级，即 4，然后任务 HighTask 进入阻塞态等待互斥信号量有效。

19.6 实验设计

19.6.1 实验内容设计

本实验设计 4 个任务：start_task、high_task、moddle_task 和 low_task，这四个任务的功能如下：

start_task：用于创建其他 3 个任务；

high_task：高优先级任务，会获取信号量，获取成功后进行相应的处理，处理完成后释放信号量；

middle_task：中优先级任务，无特殊功能；

low_task：低优先级任务，会获取信号量，获取成功后进行相应的处理，相比于高优先级任务，其占用信号量时间更长（通过软件模拟占用）。

实验首先创建二值信号量，观察优先级翻转现象；然后将二值信号量改为互斥信号量，观察优先级翻转是否得到有效抑制。

19.6.2 硬件设计

优先级翻转与互斥信号量实验硬件只需要串口，另外可以用三个 LED 灯分别表示三个任务正在运行。本实验例程中只有串口，硬件连接同第 7 章虚拟串口实验。

19.6.3 软件设计

优先级翻转与互斥信号量实验仅给出二值信号量观察优先级翻转现象的代码，实验第二部分仅需将二值信号量更换为互斥信号量，并同步更改信号量创建 API 即可。注意，二值信号量创建后需要手动设置为有效，而互斥信号量创建即有效。

由于 Arduino 的 FreeRTOS 库默认只有 4 个优先级（0-3），而 high_task 的优先级设置为 4，故需将 FreeRTOSConfig.h 中的 configMAX_PRIORITIES 宏定义更改为 8（或更高）。优先级翻转实验代码如下：

```
#include <Arduino.h>

#include <Arduino_FreeRTOS.h>

#include <semphr.h>

TaskHandle_t start_handler;

TaskHandle_t low_handler;

TaskHandle_t middle_handler;

TaskHandle_t high_handler;
```

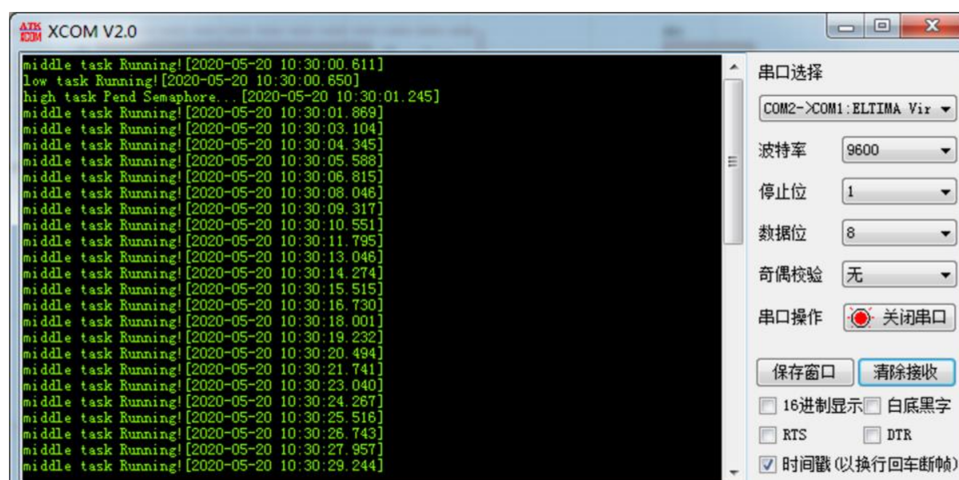
```
SemaphoreHandle_t BinarySemphr;
void start_task(void *pvParameters);
void low_task(void *pvParameters);
void middle_task(void *pvParameters);
void high_task(void *pvParameters);
void setup() {
  Serial.begin(9600);
  xTaskCreate(start_task,
    "start_task",
    128,
    NULL,
    1,
    &start_handler
  );
  vTaskStartScheduler();
}
void loop() { }
void start_task(void *pvParameters) {
  taskENTER_CRITICAL();
  // 创建二值信号量需要一次释放，若为互斥信号量不需要释放
  BinarySemphr = xSemaphoreCreateBinary();
  if(BinarySemphr) xSemaphoreGive(BinarySemphr);
  // 创建任务
  xTaskCreate(high_task,
    "high_task",
    192,
    NULL,
    4,
    &high_handler
  );
  xTaskCreate(middle_task,
    "middle_task",
    192,
    NULL,
```

```
        3,
        &middle_handler
    );
    xTaskCreate(low_task,
        "low_task",
        192,
        NULL,
        2,
        &low_handler
    );
    vTaskDelete(start_handler);
    taskEXIT_CRITICAL();
}
void high_task(void *pvParameters) {
    for(;;) {
        vTaskDelay(500/portTICK_PERIOD_MS);
        Serial.println("high task Pend Semaphore...");
        xSemaphoreTake(BinarySemphr, portMAX_DELAY);
        Serial.println("high task Running!");
        xSemaphoreGive(BinarySemphr);
        vTaskDelay(500/portTICK_PERIOD_MS);
    }
}
void middle_task(void *pvParameters) {
    for(;;) {
        Serial.println("middle task Running!");
        vTaskDelay(1000/portTICK_PERIOD_MS);
    }
}
void low_task(void *pvParameters) {
    static uint32_t times;
    for(;;) {
        xSemaphoreTake(BinarySemphr, portMAX_DELAY);
        Serial.println("low task Running!");
```

```
            // 模拟低优先级任务占用信号量
            for(times=0; times<2000000; times++) {
              taskYIELD();       // 发起任务调度
            }
            xSemaphoreGive(BinarySemphr);
            vTaskDelay(1000/portTICK_PERIOD_MS);
        }
    }
```

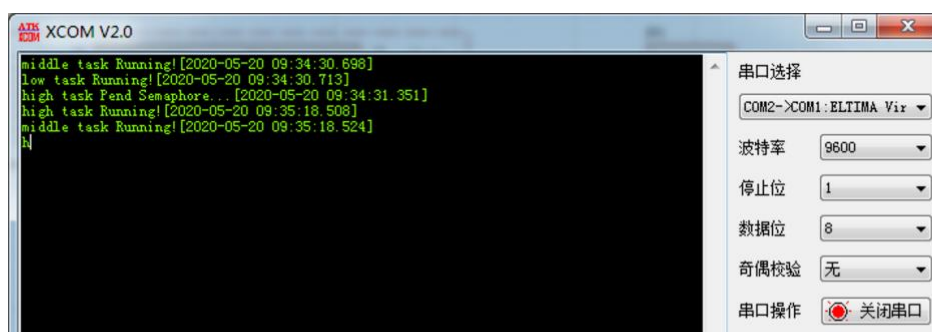
19.7 仿真测试

首先加载优先级翻转的程序文件，运行仿真。



low_task 获取到二值信号量开始运行；high_task 抢占 low_task 运行，但因 low_task 占用信号量而被挂起；此时 middle_task 优先级比 low_task 高，middle_task 一直运行，带来其优先级比 high_task 还高的错觉，即为优先级翻转；等待 low_task 完成前序工作释放信号量后，high_task 才得以运行。

然后加载互斥信号量的程序文件，运行仿真。



可以看出使用互斥信号量后，在 high_task 等待信号量时，middle_task 不会抢占 low_task 运行，因为此时 low_task 已经继承了 high_task 的优先级。

第20章 消息队列实验

20.1 消息队列简介

在讲信号量时说过，信号量实质上是消息队列。消息队列，其实就是FreeRTOS中所说的队列，是为了任务与任务、任务与中断之间的通信而准备的，可以在任务与任务、任务与中断之间传递消息，队列中可以存储有限的、大小固定的数据项目。任务与任务、任务与中断之间要交流的数据保存在队列中，叫做队列项目，队列所能保存的最大数据项目数量叫做队列的长度，创建队列的时候会指定数据项目的大小和队列的长度。队列不是属于某个特别指定的任务的，任何任务都可以向队列中发送消息，或者从队列中提取消息。

通常队列采用先进先出（FIFO）的存储缓冲机制，也就是往队列发送数据（入队）的时候永远都是发送到队列的尾部，而从队列提取数据（出队）的时候是从队列的头部提取的。但是FreeRTOS也可以使用LIFO的存储缓冲，也就是后进先出。

FreeRTOS将数据发送到队列使用数据拷贝，也就是将要发送的数据拷贝到队列中，这就意味着在队列中存储的是数据的原始值，而不是原数据的引用（即只传递数据的指针），这个也叫做值传递。相反，μC/OS的消息队列采用的是引用传递，传递的是消息指针，采用引用传递要保证消息内容一直保持可见性，也就是消息内容必须有效，那么局部变量这种可能会随时被删掉的东西就不能用来传递消息。采用值传递虽然会稍微浪费时间，但是一旦将消息发送到队列中，原始的数据缓冲区就可以删除掉或者覆写，这样的话这些缓冲区就可以被重复的使用。此外，从值传递的角度看，如果向队列发送一个消息的地址指针，那么就可以变相使用引用传递，这样当要发送的消息数据太大的时候就可以直接发送消息缓冲区的地址指针，比如在网络应用环境中，网络的数据量往往都很大的，采用数据拷贝的话就不现实。

20.2 创建消息队列

20.2.1 创建队列API函数

我们在讲信号量时提过，要使用必须要先创建，同时创建分为静态创建和动态创建，实际上信号量的创建就是调用了对应的队列创建函数。

动态创建队列的函数为 xQueueCreate()，本质上是一个宏，函数原型如下：

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, // 队列长度
                           UBaseType_t uxItemSize)    // 每项的长度
```

队列创建成功返回队列句柄，失败返回 NULL。

实际上真正创建队列的是 xQueueGenericCreate()函数，其函数原型仅比 xQueueCreate()多了一个 ucQueueType 参数，用于描述创建队列的类型，共有如下六种类型：

queueQUEUE_TYPE_BASE：普通的消息队列

queueQUEUE_TYPE_SET：队列集

queueQUEUE_TYPE_MUTEX：互斥信号量

queueQUEUE_TYPE_COUNTING_SEMAPHORE：计数型信号量

queueQUEUE_TYPE_BINARY_SEMAPHORE：二值信号量

queueQUEUE_TYPE_RECURSIVE_MUTEX：递归互斥信号量

20.2.2 队列创建过程分析

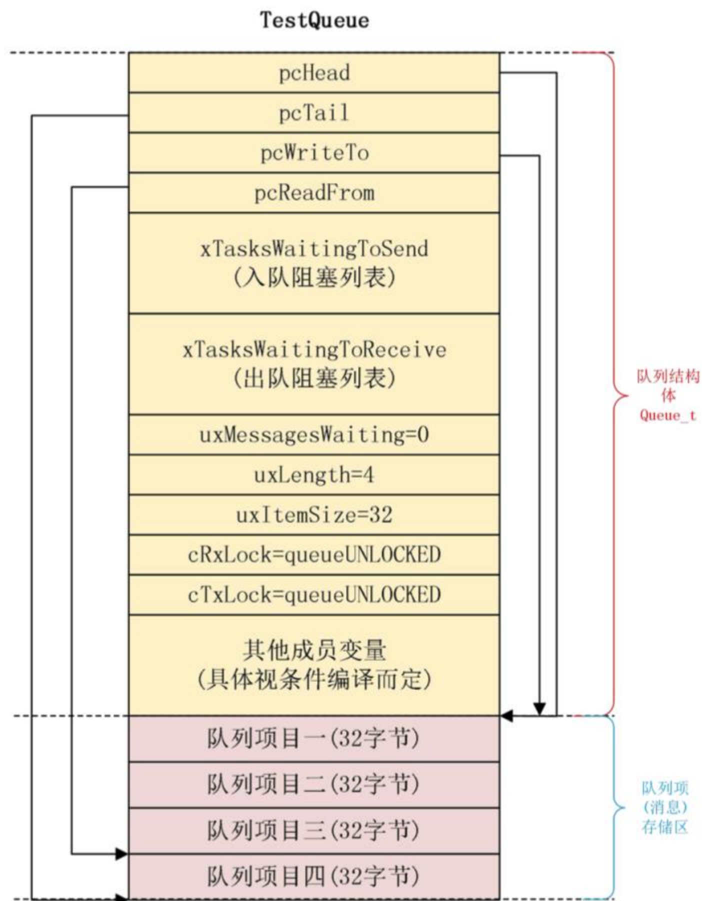
下面我们来详细分析一下 xQueueGenericCreate()函数，该函数在 queue.c 中有如下定义：

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,
                                   const UBaseType_t uxItemSize,
                                   const uint8_t ucQueueType ) {
    Queue_t *pxNewQueue;
    size_t xQueueSizeInBytes;
    uint8_t *pucQueueStorage;
    configASSERT( uxQueueLength > ( UBaseType_t ) 0 );
    // 分配足够的存储区，确保随时可以保存所有的队列项（消息）
    QueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize );
   pxNewQueue=(Queue_t*)pvPortMalloc(sizeof(Queue_t)+xQueueSizeInBytes);
    // 内存申请成功
    if( pxNewQueue != NULL ) {
        pucQueueStorage = ( uint8_t * ) pxNewQueue;
        pucQueueStorage += sizeof( Queue_t );
        #if( configSUPPORT_STATIC_ALLOCATION == 1 ) {
            // 队列由动态方法创建，将字段标记为 pdFALSE
            pxNewQueue->ucStaticallyAllocated = pdFALSE;
        }
```

```
        #endif
        prvInitialiseNewQueue( uxQueueLength, uxItemSize,
                               pucQueueStorage, ucQueueType, pxNewQueue );
    }
    else {
        traceQUEUE_CREATE_FAILED( ucQueueType );
        mtCOVERAGE_TEST_MARKER();
    }
    return pxNewQueue;
}
```

可以看出函数 xQueueGenericCreate()重要的工作就是给队列分配内存，当内存分配成功以后调用函数 prvInitialiseNewQueue()来初始化队列，在队列初始化中又调用函数 xQueueGenericReset()复位队列。关于这两个函数可以自行查看函数定义。

比如创建一个有 4 个队列项、每个队列项长度为 32 字节的队列 TestQueue，创建成功后的队列如图所示：



20.3 向队列发送消息（入队）

向队列发送消息的 API 函数比较复杂，如下表所示：

分类	函数	描述
任务级入队函数	xQueueSend()	发送消息到队列尾部（后向入队），这两个函数是一样的
	xQueueSendToBack()	
	xQueueSendToFront()	发送消息到队列头（前向入队）。
	xQueueOverwrite()	发送消息到队列，带覆写功能，当队列满了以后自动覆盖掉旧的消息
中断级入队函数	xQueueSendFromISR()	发送消息到队列尾（后向入队），这两个函数是一样的，用于中断服务函数
	xQueueSendToBackFromISR()	
	xQueueSendToFrontFromISR()	发送消息到队列头（前向入队），用于中断服务函数
	xQueueOverwriteFromISR()	发送消息到队列，带覆写功能，当队列满了以后自动覆盖掉旧的消息，用于中断服务函数

此处只讲解任务级入队函数。

1、函数 xQueueSend()、xQueueSendToBack()、xQueueSendToFront()

这三个函数都是用于向队列中发送消息的，本质都是宏，其中函数 xQueueSend()和 xQueueSendToBack()是一样的，都是后向入队，即将新的消息插入到队列的后面。函数 xQueueSendToToFront()是前向入队，即将新消息插入到队列的前面。这三个函数的参数和返回值都是一样的，如下：

参数：

- xQueue，队列句柄，指明要向哪个队列发送数据；
- pvItemToQueue，指向要发送的消息，发送时将该消息拷贝到队列；
- xTicksToWait，阻塞时间，即当队列满时任务阻塞等待队列的时间。

队列发送成功返回 pdPASS，失败返回对应的 error code。

2、函数 xQueueOverwrite()

此函数也是用于向队列发送数据的，当队列满了以后会覆写掉旧的数据，不管这个旧数据有没有被其他任务或中断取走。这个函数常用于向那些长度为 1 的队列发送消息，此函数也是一个宏。此函数的参数与上述三个函数类似，但没有 xTicksToWait，因为队列满了就覆盖重写，不存在失败和阻塞。

3、函数 xQueueGenericSend()

此函数才是真正完成向队列发送消息的函数，上述四个任务级的入队函数均调用此函数。此函数参数增加一个 xCopyPosition，上述四个函数均通

过配置此参数来决定使用哪种方式入队。此函数的定义在 queue.c 中，可以自行学习其具体过程。

20.4 从队列读取消息（出队）

有入队就有出队，从队列读取消息的 API 函数如下表所示：

分类	函数	描述
任务级 出队函数	xQueueReceive()	从队列中读取队列项，并且读取完以后删除掉队列项
	xQueuePeek()	从队列中读取队列项，并且读取完以后不删除队列项
中断级 出队函数	xQueueReceiveFromISR()	从队列中读取队列项，并且读取完以后删除掉队列项，用于中断服务函数中
	xQueuePeekFromISR ()	从队列中读取队列项，并且读取完以后不删除队列项，用于中断服务函数中

此处只讲解任务级出队函数。

1、函数 xQueueReceive()、xQueuePeek()

这两个函数用于在任务中从队列中读取一条消息，不同的是读取成功后 xQueueReceive()会将这条消息从队列中删除，而 xQueuePeek()不会将消息删除。由于 FreeRTOS 的队列采用数据拷贝，所以我们需要提供一个数组或缓冲区来保存读取到的数据，所读取的数据长度是创建队列时所设定的每个队列项目的长度。这两个函数的参数和返回值都是一样的，如下：

参数：

xQueue，队列句柄，指明要读取哪个队列的数据；

pvBuffer，保存数据的缓冲区；

xTicksToWait，阻塞时间，即当队列空时任务阻塞等待队列的时间。

从队列中读取消息成功返回 pdTRUE，失败返回 pdFALSE。

2、函数 xQueueGenericReceive()

此函数才是真正完成从队列读取消息的函数，上述两个任务级的出队函数均调用此函数。此函数增加一个参数 xJustPeek，用于标记读取消息成功后是否删除该队列项。

出队函数的具体过程和入队函数类似，可以自行对照着源码学习。

20.5 实验设计

20.5.1 实验内容设计

本实验设计 4 个任务：TaskSerial、TaskADCPin0、TaskADCPin1 和

TaskBlink，这四个任务的功能如下：

TaskSerial：从队列中读取消息并通过串口输出；

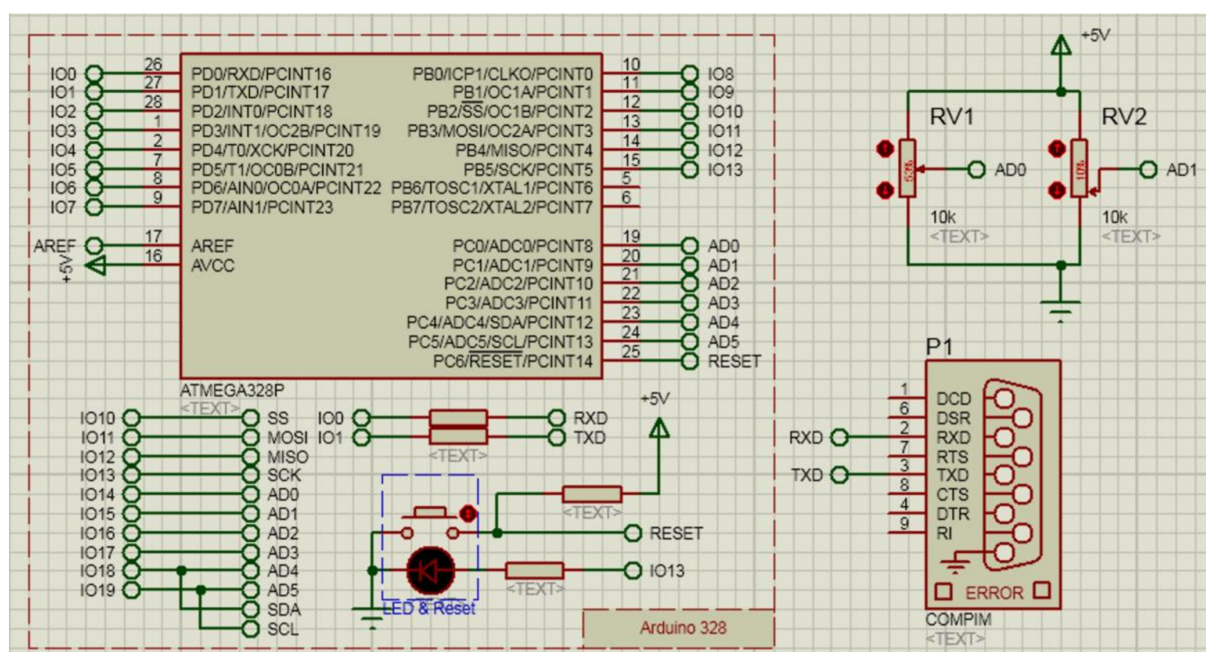
TaskADCPin0：读取 ADC0 的数据并发送到队列；

TaskADCPin1：读取 ADC1 的数据并发送到队列；

TaskBlink：板载 LED 灯闪烁，表明系统正常运行。

20.5.2 硬件设计

由于需要从模拟端口读取数据，为了便于实验演示，对两个 ADC 接口分别设计一个分压电路，通过电位器改变电压值。电位器选用 POT-HG，可以按百分比调节电位器阻值。实验硬件连接如下：



20.5.3 软件设计

队列项设计为一个结构体，结构体成员包括 ADC 端口名和 ADC 数值。消息队列实验代码如下：

```
#include <Arduino.h>
#include <Arduino_FreeRTOS.h>
#include <queue.h>
struct pinRead {
  int pin;
  int value;
};
```

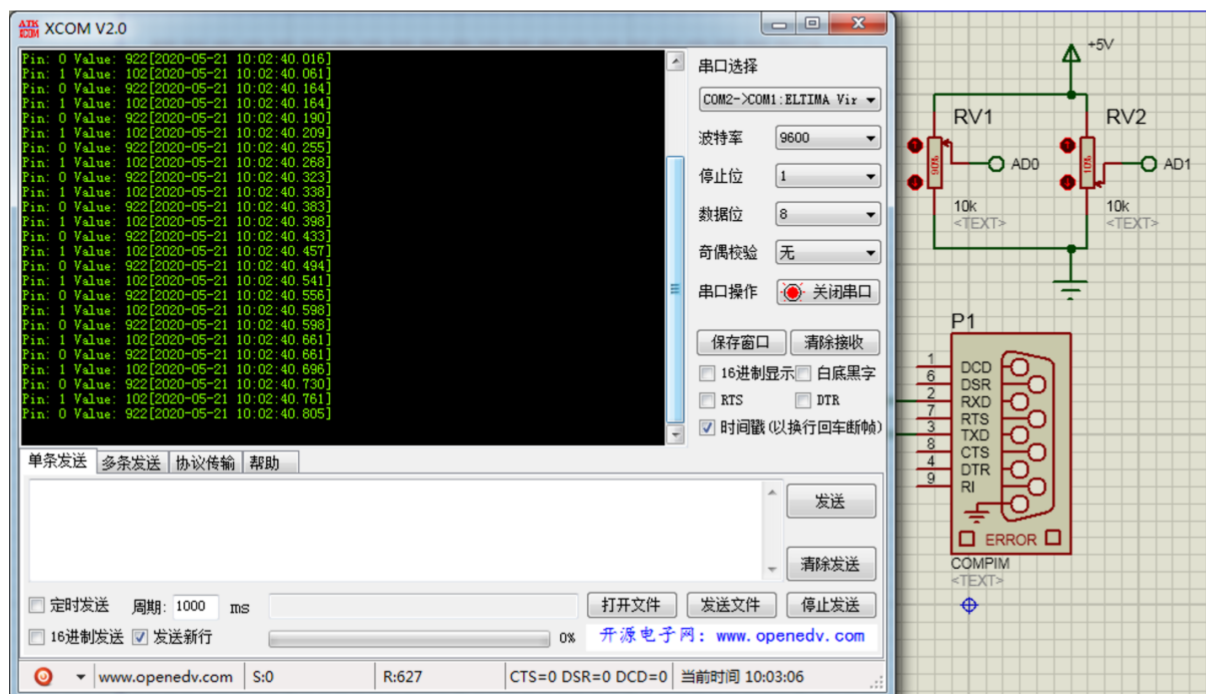
```
QueueHandle_t structQueue;
TaskHandle_t serial_handler;
TaskHandle_t adc0_handler;
TaskHandle_t adc1_handler;
TaskHandle_t blink_handler;
void TaskSerial(void* pvParameters);
void TaskADCPin0(void* pvParameters);
void TaskADCPin1(void* pvParameters);
void TaskBlink(void* pvParameters);
void setup() {
  structQueue = xQueueCreate(10, sizeof(struct pinRead));
  if (structQueue != NULL) {
    xTaskCreate(TaskSerial,
                "Serial",
                128,
                NULL,
                2,
                &serial_handler);
    xTaskCreate(TaskADCPin0,
                "ADCPin0",
                128,
                NULL,
                1,
                &adc0_handler);
    xTaskCreate(TaskADCPin1,
                "ADCPin1",
                128,
                NULL,
                1,
                &adc1_handler);
  }
  xTaskCreate(TaskBlink,
              "Blink",
              128,
```

```
                NULL,
                0,
                &blink_handler);
}
void loop() {}
void TaskADCPin0(void *pvParameters) {
  struct pinRead currentPinRead;
  for (;;) {
    currentPinRead.pin = 0;
    currentPinRead.value = analogRead(A0);
    xQueueSend(structQueue, &currentPinRead, portMAX_DELAY);
    vTaskDelay(50 / portTICK_PERIOD_MS);
  }
}
void TaskADCPin1(void *pvParameters) {
  struct pinRead currentPinRead;
  for (;;) {
    currentPinRead.pin = 1;
    currentPinRead.value = analogRead(A1);
    xQueueSend(structQueue, &currentPinRead, portMAX_DELAY);
    vTaskDelay(50 / portTICK_PERIOD_MS);
  }
}
void TaskSerial(void * pvParameters) {
  Serial.begin(9600);
  struct pinRead currentPinRead;
  for (;;) {
    if (xQueueReceive(structQueue, &currentPinRead, portMAX_DELAY)
                                                              == pdPASS) {
      Serial.print("Pin: ");
      Serial.print(currentPinRead.pin);
      Serial.print(" Value: ");
      Serial.println(currentPinRead.value);
    }
```

```
    }
}
void TaskBlink(void *pvParameters) {
  pinMode(LED_BUILTIN, OUTPUT);
  for (;;)
  {
    digitalWrite(LED_BUILTIN, HIGH);
    vTaskDelay( 250 / portTICK_PERIOD_MS );
    digitalWrite(LED_BUILTIN, LOW);
    vTaskDelay( 250 / portTICK_PERIOD_MS );
  }
}
```

20.6 仿真测试

运行仿真，通过拖动电位器指针或点击电位器的上下键可以改变电位器阻值。比如设置两个电位器分别为 10%和 90%接入电阻，则仿真结果如下：



Arduino Uno 的 ADC 参考电压默认是 5V，ADC 为 10 位，输出 ADC 值一个为 102，一个为 922。同时板载 LED 闪烁。

第五篇 综合实验篇

第21章 智能家居照明控制系统实验